DIVISION OF COMPUTER SCIENCE

Towards a Satisfaction Relation between CCS Specifications and their Refinements

Technical Report No. 152

Elizabeth Jean Baillie

April 1992

TOWARDS A SATISFACTION RELATION BETWEEN CCS SPECIFICATIONS AND THEIR REFINEMENTS

ELIZABETH JEAN BAILLIE

A thesis submitted in partial fulfilment of the
requirements of the University of Hertfordshire
for the degree of Doctor of Philosophy

April 1992

This research programme was carried out
in collaboration with Praxis Systems plc

Contents

List of Figures

Towards a Satisfaction Relation between CCS Specifications and their Refinements

Elizabeth Jean Baillie

Abstract

The thesis is concerned with the application and applicability of CCS and, in particular, the problem of establishing a satisfaction relation between specifications and their refinements in CCS. The problems encountered arise from the instability which in general results when agents are composed and certain actions restricted. Bisimulation proves to be elusive in the presence of leading τ's in an expansion. *Testing* equivalence, the conjunction of *may* and *must* equivalences, is investigated. *may* testing is unaffected by either divergence or internal nondeterminism; *must* testing is affected by both. This is similarly hard to establish.

In the absence of equivalence we investigate testing *preorders* between specifications. We use the example of a level crossing to test and illustrate the ideas developed. We specify the required behaviour of the crossing when viewed as a single entity. The specification is refined by decomposition into its lower level components, which are then composed. We use the expansion law of CCS in conjunction with the Concurrency Workbench to analyse the behaviour of the composed system.

What we find is that *may* equivalence between a specifications and its refinements can be achieved quite easily, though in general the model is intrinsically too weak to prove all required properties of a system. Successive refinements of the specification are related by the *must* preorder but in a counter-intuitive way, often entailing loss of *liveness*. In general, there is no bisimulation between refinements.

In view of this we propose a conservative extension to CCS, called Concurrent CCS, which sets out to remove what seems to be arbitrary nondeterminism in composed systems. It provides a new action prefixing operator which 'ties together' its operands, and a concurrent composition operator. The result is greater stability in the behaviours of composed systems, though at a price. We prove a Concurrent Expansion Law for the extended calculus and suggest some possible notions of equivalence.

Acknowledgements

I would like to express my thanks to several people: first of all to David Smith, my Director of Studies, for all the help and support he has given me, and particularly for the stimulus provided by his paper on Concurrent CCS [Smi91]: secondly, to Bob Dickerson for his unique Unix rescue and support service, the value of which it would be hard to overstate, and for his helpful comments on the early drafts. Then in alphabetical order: Rod Adams for his patience and generous support: Ruth Barrett for raising my consciousness of the level crossing: Bean and Gordon for valued friendship and timely encouragement: Bruce Christianson for many helpful and stimulating discussions: Laurence Dixon and Steve Stott for their kind encouragement and support: David Fensome for his efforts at project-managing the unmanageable and his contribution to the paper from which chapter 3 is taken: Bob Lawn, Paul Morris, and the technician staff for invaluable technical support: Wilf Nicholls for his careful reading of chapter 2 and the many helpful comments he made on it: and Ben Potter for his help in proof reading and the insight of his experience in industrial software engineering.

The work was supported by a grant from the Science and Engineering Research Council, to whom I am grateful.

Chapter 1

Introduction

The need for a formal engineering discipline in the specification and design of software is becoming increasingly recognized. As automated systems grow and proliferate, the consequences of error and failure increase in cost, and not only in terms of time and money. The object of applying formal methods at the early stages of software development is to introduce such an engineering discipline through the use of mathematical and logical tools. These provide a precise notation for the expression of system requirements and their refinement and a deductive mechanism for reasoning about these formalized specification and design statements.

Typically, system requirements are expressed in natural language by the customer and then processed to produce a specification, also heavily dependent on natural language. Experience has shown that it is at these early stages of the development that errors are most likely to occur and at the later stages when they are most likely to manifest themselves; to quote Ince [Inc88]:

> ...those notations which are most heavily dependent on natural language are those which occur at the beginning of the software life-cycle where undetected errors can lead to massive overruns or even total cancellation.

This leads to the view that investment is needed in the early stages of development to ensure that the customer's requirements are fully understood and precisely specified.

1.1 Safety-Critical Systems

It is in the area of safety-critical systems that the importance of correct and verifiable software is most clearly recognized; we are only too well aware of the consequences of software failure in civil aircraft, nuclear reactors and so on. The MOD draft document Def Stan 00-55 [UK 91] sets out proposals for the use of formal methods in the development of safety critical systems, which it defines as those whose failure may result in loss of life. The standard is primarily concerned with the procurement of defence software, but the concerns it expresses and the problems it is seeking to address have relevance for a much wider field. Safety-critical systems are often concurrent, communicating, real-time systems of daunting complexity and there are no easy solutions to the problems they present. Nevertheless, these problems must be addressed in software engineering as in other, more traditional, branches of engineering. The fact that the perception of software differs from that of, say, civil engineering projects should not reduce the manufacturer's degree of responsibility for producing a fully engineered quality product. Funds are being made available for research into the development of formal methods for safety-critical systems with a view to providing some practical results.

1.2 Specification

There are a number of formal specification and design methods available. Some are targeted at specific problem areas, others are intended as more general-purpose tools. The languages fall into three broad (though not necessarily mutually exclusive) categories:

- algebraic, e.g. OBJ, CLEAR, used particularly for specifying abstract data types:
- model-based, typed set-theoretic, e.g. Z, VDM, the most widely used general-purpose languages:
- process-based, e.g. CCS, CSP, designed to specify concurrent, communicating systems.

Other languages such as LOTOS and RAISE aim to combine aspects of two or more of the above, but it is usually found that the heightened expressive power of such languages results in a lowering of analytical power. There are strong arguments on both sides of this seemingly inevitable trade-off.

1.2.1 Concurrency

There are a number of well-established techniques in the literature for specifying concurrent systems. For example, in Hoare's CSP [Hoa85] a specification is a predicate over traces; in CCS [Mil80, Mil89] a specification is usually a term, or family of terms, in the algebra, though it may be expressed as a collection of modal formulae [HM85]; Lamport [Lam83] suggests that concurrent systems can be specified in terms of their safety and liveness requirements. Temporal and modal logics can also be used to describe required behaviours. These methods are by no means mutually exclusive; one may be implicit in another. For example, unsafe time orderings of events can be implicitly excluded in a CCS specification; illegal or unsafe traces can be prohibited in CSP; and the modal operator $\langle\rangle$ together with the derived [] provide an expressive language for stating both safety and liveness requirements. What we need to remember when using any of these techniques, whether for concurrent or sequential systems, is that we are building models of an intuitive idea about the behaviour of a real system; that is, we are at least two steps removed from the real world. When we prove something about a model we are not necessarily any nearer guaranteeing that the physical running system will always behave in the same manner. This is not an argument against building mathematical models; but we need to be aware of the limitations of formal techniques as well as their benefits.

1.3 Refinement

Ideally, the engineering discipline should be a feature of the whole development process, from the first question *what?* to the last question *how?*. Having specified a system formally, we need to be able to proceed towards implementation through design steps which preserve correctness in some sense; we should like to be able to

establish a satisfaction relation between discrete steps of the development. Carroll Morgan's Refinement Calculus [Mor90] and the rules and proof techniques for refining (or *reifying*) data in VDM [Jon86] are examples of techniques which have been developed to address this particular question.

Refining the behaviour of concurrent processes is particularly complex. In addition to the problems that arise in sequential systems (which include the possibility of non-termination) we also have to consider such issues as inter-process communication and the consequent potential for internal nondeterminism and divergence. We need to think very hard about what we mean when we say that a concurrent system 'satisfies' its specification. It is not appropriate to think of concurrent systems as simple mappings from input to output states, as we might think of sequential systems. We need to ask such questions as: how much internal nondeterminism we are prepared to tolerate: how far we should be concerned about the possible divergent behaviour of one or more components of the system; whether we are interested in the inner workings of the system or just wish to view it as a black box and consider only its externally visible behaviour: whether timing considerations are critical: whether there are safety and liveness issues that need be to guaranteed: and all these questions on top of the problems of data correctness.

In addition, there is the problem referred to by Medawar as *emergence*; in his series of lectures entitled 'Induction and Intuition in Scientific Thought' [Med69] he says

> Each tier of the ...hierarchy makes use of notions peculiar to itself.... In each plane or tier of the hierarchy new notions or ideas seem to emerge that are inexplicable in the language or with the conceptual resources of the tier below.

Medawar is here discussing the relationship of sociology to biology in the hierarchical structure of Nature, but something similar is true in the development of software. Programs run on finite machines whose limitations are very much the business of the programmer but which may hardly impinge on the customer's consciousness. How far such issues are allowed to intrude into higher levels of specification, or, indeed, whether they are in fact 'inexplicable in the tier below' (for 'below' read 'above'!) is

not at all clear. Notwithstanding, they need to be taken into account in the process of establishing the correctness of an implemented specification.

1.4 Overview of the thesis

To quote Milner [Mil89]:

> People will use it [CCS] only if it enlightens their design and analysis of systems; therefore the experiment is to determine the extent to which it is *useful*, the extent to which the design process and analytic methods are indeed improved by the theory...the degree of this usefulness is hard...to predict from the mathematical nature of the theory.

The concern of the thesis is with the application and usefulness of CCS to the software development process: in particular, with the problem of establishing a satisfaction relation between specifications and their refinements in CCS, an important aspect of the usefulness of the theory. It should be said that by 'refinement' here, we mean something very specific, namely, the decomposition of the specification into the physical objects which go to make up the system. The process of identifying and specifying these objects is referred to throughout as 'agent decomposition', the term 'refinement' having acquired a connotation which might mislead in this case (though we feel it properly describes the process).

We are considering the use of CCS in the early stages of the life-cycle, i.e., at a high level of abstraction. We use the words 'specification' and 'design' interchangeably; arguably, every level of abstraction is a specification of the level below and a design of the level above—and every design must meet its specification. The problems encountered in finding a satisfaction relation between the two in CCS arise from the internal nondeterminism—in the form of the special action τ—which in general arises when agents are composed under the Expansion Law and certain actions restricted. Bisimulation, the CCS standard notion of equivalence, proves to be elusive in the presence of leading τ's in an expansion. *Testing* equivalence, due to de Nicola and Hennessy [dNH84], is the conjunction of two distinct equivalences referred to as *may* and *must*

equivalence. These depend upon the outcome of tests or experiments to which the process may be subjected; testing equivalence is similarly hard to establish—or, at least, the *must* part of the equivalence. *may* equivalence, which coincides with the *trace* model of CSP [Hoa85], can usually be proved, but for most practical purposes this on its own is too weak to prove all the required properties of a system. Testing is not a standard notion of CCS but is usefully implemented in the Concurrency Workbench, the software tool used throughout to construct equivalences and other relations.

In the absence of equivalence, then, we need to ask what relationship between specifications we are prepared to regard as demonstrating satisfaction. There are three main candidates, all preorders: *prebisimulation*, which orders processes on the basis of their potential for divergent behaviour (convergent processes which are related by prebisimulation are, in fact, bisimilar): and the two testing preorders, referred to as *may* and *must* testing. Prebisimulation does not address the problem of internal nondeterminism. As has already been stated, the *may* preorder depends only upon the language, or traces, of a process; it is unaffected by either divergence or internal nondeterminism. The *must* preorder, on the other hand, is affected by both of these.

We have used the example of a level crossing to test and illustrate the ideas developed; it was chosen as being a safety critical system which is small enough to be modelled without too much complexity but which nevertheless displays all the features we wish to examine; and the importance of safety and liveness in such a system are clear. We first of all specify at a very high level of abstraction the required behaviour of the crossing when viewed as a single entity. We then identify the lower level components of the system and specify these also. We use the expansion law of CCS in conjunction with the concurrency workbench to analyse the behaviour of the composed system. We can continue this process through as many stages as we wish, decomposing the higher level components into more concrete objects, composing them within the calculus and comparing the behaviour of each stage with its predecessor for equivalence or preordering.

1.4.1 Preordering

What we find is that *may* equivalence between successive decompositions can be achieved quite easily. We may regard this as establishing *safety*; the system at every stage of decomposition is capable of the same sequences of actions as the level above and, ultimately, as the top level specification. So if unsafe traces are prohibited in a specification and *may* equivalence with the design has been shown to hold, we can be sure they are also prohibited in the design. In general, however, neither bisimulation nor *must* equivalence can be established; both are precluded by the presence of leading τ's in the behaviour expressions of the composed systems. What we can demonstrate without too much difficulty is the *must* preorder, rather than equivalence, but this often means that we lose properties of liveness enjoyed by the specification.

The testing preorders relate successive decompositions in what seems to be a counter-intuitive way; what we might expect as we add detail to the design is that it should become more and more deterministic. This is the case in Morgan's Refinement Calculus; the chain of specifications, or 'programs' as they are all called, has the specification at the bottom[1] and the final implementation at the top. What we find in CCS, however, is that our chain has the specification at the top and the implementation at the bottom. This is due to the internal nondeterminism which increases with decomposition despite the fact that we are being more and more concrete in our designs. This is inevitable in this model since nondeterminism is the basis of the ordering and it is hard to avoid introducing it in the process of decomposition in CCS; but it is unsatisfactory.

1.4.2 Concurrent CCS

In view of this, then, we propose a conservative extension to CCS, called Concurrent CCS [Smi91], which sets out to remove what seem to be the arbitrary and misleading cases of nondeterminism. All τ's look alike, whether they arise from arbitrary system choice or from responsible signalling mechanisms which we might regard as anything but nondeterministic. It is the problems arising from this form of nondeterminism

[1] This refers to 'starting point' of the chain under consideration—not to be confused with $\perp$

which Concurrent CCS seeks to address. It provides a new action prefixing operator which allows its operands to run in true concurrency; the result is greater stability in the behaviours of composed systems, though at a price. We prove a Concurrent Expansion Law for the extended calculus and suggest some possible notions of equivalence.

1.5 Summary

Chapter 2 outlines the theoretical background to the calculus. It begins with an informal look at process modelling, and in particular, how nondeterminism is modelled in CCS and the algebra of Hennessy, which we refer to as ATP. We then go on to look at the three principal ways in which the signatures for process algebras can be given formal semantics: operational, denotational and axiomatic. In chapter 3 we examine a fairly lengthy case study which identifies some of the problems we may expect to encounter when attempting to prove CCS designs equivalent to their specifications. It becomes clear from this case study that we need to look again at our notion of equivalence and ask ourselves exactly what we are trying to model. Chapter 4 examines some of these ideas, in particular, bisimulation and testing equivalence. In chapter 5 we take a second, much simpler, case study and use it as a testbed for the ideas considered in chapter 4. We develop a satisfaction relation between CCS specifications and their decomposition refinements which we believe is, in general, the best that can be established, though it is not satisfactory. In chapter 6 we introduce Concurrent CCS with a few examples of its application. We propose and later prove the Concurrent Expansion Law. We then reconsider the case study of chapter 5 in the light of the extended calculus and re-specify it making use of the new operators. We consider some candidates for an equivalence over CCCS agents. In chapter 7 we draw conclusions and suggest areas for further work.

Chapter 2

Theoretical Background

The seminal works of Milner ([Mil80, Mil89]) and Hoare ([Hoa85]) laid the foundations for what have come to be known as process algebras. Although it is sometimes convenient to classify formal specification languages into either algebraic, model-based or process-based, in fact, process algebras fall into all three categories. They provide initial semantics (that is, equality is subject to proof) for a signature whose operators are, typically, action-prefixing, choice, restriction, relabelling and composition. Processes (or agents), which may be finite or recursive, are modelled as nondeterministic choices between sequences of atomic actions. Choice may be internal (made by the system) or external (made by the environment), different algebras taking differing views of internal nondeterminism. Interleaving semantics for concurrency are provided in general; true concurrency is not modelled, though work is being done in this area [CH89a].

We begin by taking an informal look at how we can capture the behaviours of processes in an algebraic notation. We see how different models of internal nondeterminism describe quite different behaviours. In section 1, we introduce two theories in which these behaviours are formalized, those of Milner [Mil80, Mil89] and Hennessy [Hen88]. In section 2 we look at three different semantics for the languages of processes, namely, operational, denotational and axiomatic. In the last section we discuss briefly some of the notions of refinement in the literature.

2.1 Process Modelling

We need to begin by asking ourselves what we are trying to model. It is not appropriate to regard concurrent systems as mappings from input to output states; instead, we model the *behaviours* of systems in a way that takes account of issues such as action sequencing, the choices the system may present to the user, the 'choices' made by the system itself, concurrency and interprocess communication, and recursion. We discuss CCS and Hennessy's algebra (which we shall refer to as ATP, for Algebraic Theory of Processes, [Hen88]). We will motivate our consideration of these issues with some toy examples. A glossary of symbols is given in Appendix A. By convention, CCS agent identifiers use upper case letters and ATP, lower case. ATP also omits '.', the prefix operator.

2.1.1 Internal and External Nondeterminism

We begin with the example of an airport runway; modelled from the point of view of the controller, we can say that a runway permits planes *either* to take off *or* to land, and the controller may choose. We can express this choice as

$$Runway1 \stackrel{def}{=} takeoff.0 + land.0$$

This is a very expensive runway which may only be used once; the 0 after each action indicates the inability of the runway to engage in further action. The + indicates that a choice may be made by an observer who is outside the system (in this case, the controller). We can specify a more realistic and much cheaper runway which may be used indefinitely for the same purpose:

$$Runway2 \stackrel{def}{=} takeoff.Runway2 + land.Runway2$$

We shall consider recursion in more detail later; for the moment we concentrate on the modelling of choice and action sequencing. We can see from this specification that the runway is safe in that taking off and landing cannot both occur together. However, if we take a wider view of the landing area we can see that planes need to taxi to and from the landing strip itself and we might want to incorporate this part

of the process in the specification. So we have

$$Runway3 \stackrel{def}{=} taxi.takeoff.Runway3 + land.taxi.Runway3$$

What we now have is as follows: if a plane on the ground starts to taxi towards the runway, it must be allowed to take off before anything can be allowed to land: if, on the other hand, a plane has landed then it must be allowed to taxi (back to the terminal building) before a plane on the ground can begin to move. The choice of the first action in a sequence locks the system into that sequence, which must then be completed before the system can return to its original (or any other) state. Both CCS and ATP model sequencing and external choice in this way.

We now consider the same runway as seen by an observer standing in the spectators' gallery. He has no control over the system but will be happy just to wait and see what happens. In ATP this choice is expressed

$$Runway4 \stackrel{def}{=} taxi.takeoff.Runway4 \oplus land.taxi.Runway4$$

where $\oplus$ represents internal nondeterminism. Both sequences are visible to the observer but he cannot make the choice; it is made in a manner which is opaque to him. In CCS this is represented

$$Runway5 \stackrel{def}{=} \tau.taxi.takeoff.Runway5 + \tau.land.taxi.Runway5$$

CCS provides only one choice operator. The nature of the choice is determined by the structure of its operands. The action τ is not visible but nevertheless locks the system into the particular sequence which it leads. $Runway5$ is *unstable*, i.e., its expression contains leading τ actions; its behaviour cannot be predicted. These different models of nondeterminism have profound consequences for their respective algebras; they give rise to different notions of equivalence arising from different intuitions about the behaviours of larger systems, as we shall see.

We now consider a soft drinks vending machine which serves orange, lemon or cola. The cans of orange have been loaded correctly but the assistant was distracted while stacking the lemon and it got mixed up with the cola. In ATP we can write this as

$$VM1 \stackrel{def}{=} 50p.(orange.VM1 + (lemon.VM1 \oplus cola.VM1))$$

$VM1$ takes the customer's $50p$ then presents him with the choice of orange (which he may always choose and be sure of getting) or else one of lemon or cola, 'chosen' by the machine; the scope of $\oplus$ is its immediate operands. In CCS we cannot model this situation directly; if we were to write

$$VM2 \stackrel{def}{=} 50p.(orange.VM2 + (\tau.lemon.VM2 + \tau.cola.VM2))$$

the effect of the leading τ's is to propagate the nondeterminism through the whole expression. The instability of *lemon* and *cola* affects the choice of *orange* as well; there is no way of predicting what we will get. We *can* model $VM1$ in CCS as follows:

$$VM1' \stackrel{def}{=} 50p.(\tau.(orange.VM1' + lemon.VM1') + \tau.(orange.VM1' + cola.VM1'))$$

We exploit the fact that we cannot restrict the scope of τ; *orange* is an option in each summand of $VM1'$. But this is not a natural way to specify such a system; it does not model our intuition about the machine's behaviour (although its semantics are the same as $VM1$'s).

2.1.2 Concurrency and Communication

We return to the example of the runway and model the behaviour of the air-traffic controller. He may send the *ok* signal either to the plane waiting to take off or to the one waiting to land, and he can choose which to send.

$$Controller \stackrel{def}{=} \overline{ok_t}.Controller + \overline{ok_l}.Controller$$

The overline, by convention, indicates an output signal. For $Runway6$ to communicate with $Controller$ and admit one of these signals, it must contain complementary actions appropriately placed:

$$Runway6 \stackrel{def}{=} ok_t.taxi.takeoff.Runway6 + ok_l.land.taxi.Runway6$$

CCS provides a composition operator denoted by | which allows its operands to run *concurrently*, by which we mean that the actions of the operands may interleave arbitrarily, subject to the structure of each agent. The behaviour of composed agents is governed by the Expansion Law of CCS whose use is demonstrated in later chapters;

it enables concurrently composed agents to be expressed as a choice between sequences of actions. (ATP has a corresponding equational law for the expansion of its composed processes). We can *restrict* certain actions, that is, prevent them communicating visibly; in this case, they may only communicate internally with their respective complements, resulting in the special action τ. When τ leads a sequence of actions in an expression the resulting agent is unstable. In the above example it is natural to restrict by the set $\{ok_t, ok_l\}$; we do not wish to send these signals to the environment. So we have

$$\begin{aligned} Airport &\stackrel{def}{=} Runway6|Controller\backslash\{ok_t, ok_l\} \\ &= \tau.taxi.takeoff.Airport + \tau.land.taxi.Airport \end{aligned}$$

We note in passing that this is the behaviour of $Runway3$, the spectators' model.

2.1.3 Recursive Processes

We return to $Runway2$ above which is defined recursively:

$$Runway2 \stackrel{def}{=} takeoff.Runway2 + land.Runway2$$

The observer tracing the behaviour of the runway and taking notes might have recorded something like

$$takeoff.land.land.takeoff.takeoff.takeoff \ldots$$

The intuition that $Runway2$ seeks to capture is that planes may either take off or land, *ad infinitum*. We look at the formal interpretation of recursively defined processes in 2.2.3.

2.2 Formal Semantics

2.2.1 Signatures

The language constructs of CCS consist of a set Act of actions (including complementary actions and the distinguished action τ), and operations, or combinators, as follows: action prefixing (denoted by . which is commonly omitted)), choice (+),

composition (|), restriction (\) and relabelling([f], where f is a relabelling function). The constant agent NIL, or 0, is defined to be the agent which can engage in no action at all. ATP's signature is similar except that there is no action to correspond to τ; internal nondeterminism is modelled by the binary operation $\oplus$. Also, actions are modelled by unary functions. The last point does not result in a difference in semantics, though the operation $\oplus$ does, as we have already noted. ATP's signature also contains a distinguished symbol Ω[1], a function of arity 0, representing the undefined process. We return to this in later sections. From the signatures we can generate terms in each algebra by applying the appropriate grammar rules to give the *term algebra* [Bai90]. Strictly speaking, these terms are purely syntactic and simply give us a language for processes. The semantics are initial; agents are assumed to be distinct unless and until they can be shown to be equivalent (by some definition of equivalence).

2.2.2 Operational Semantics

The semantics of CCS is given as a labelled transition system (see Appendix A). So, for instance, the agent $a.NIL$ may engage in the action a and then evolve to NIL; more formally, $a.NIL \xrightarrow{a} NIL$, or in general, $a.P \xrightarrow{a} P$ for any process P. The same transition relation is applied to the composition of agents such as $P \xrightarrow{a} P'$ and $Q \xrightarrow{\overline{a}} Q'$; we then have $P|Q \xrightarrow{\tau} P'|Q'$, meaning that the composition of P and Q evolves unobserved into the state $P'|Q'$. Hennessy defines a separate notation for such silent transitions; so in ATP, $p|q \rightarrowtail p'|q'$, and $p \oplus q \rightarrowtail p$, $p \oplus q \rightarrowtail q$.

We can regard the CCS agent whose expression is $\tau.a + \tau.b$ as analogous to ATP's $a \oplus b$; but for the CCS agent $P \stackrel{def}{=} a + \tau.b$, the ATP equivalent is less obvious. Informally, this process will always allow action b but may sometimes pre-empt a; we can express this in ATP as $p \stackrel{def}{=} b + (a \oplus b)$. The operator $\oplus$ does not affect the first occurrence of b in the expression; but if we want action a, we must take our chances with the system, which may 'decide' to refuse and only allow b. Such 'translations' between CCS and ATP are often, though not always possible.

[1] Ω can be defined in within the constructs of each algebra. Hennessy defines Ω to be the process given by $rec\,x.x$; Milner describes an agent which he calls *Infinite Chatter*, defined as $IC \stackrel{def}{=} \tau.IC$

Labelled transition semantics are the standard model for CCS and give rise to a behavioural equivalence over agents, namely, *bisimulation.* This is discussed more fully in chapter 4; it is an elegant theory which has been automated in the Concurrency Workbench (also discussed in chapter 4). Hennessy defines two operational equivalences—the kernels of two preorders—based on the tests that a process *may* or *must* pass. The tests have a similar syntax to the processes being tested, except for two differences; first of all, so that the tester or experimenter may have a greater degree of control over the tests than the process being tested, he defines a special action 1, representing a 'transition' made by the experimenter without the process's cooperation; and secondly, since we need to distinguish between successful and unsuccessful tests, he defines an action w by which the experimenter reports success. The interaction of a process p and an experimenter e is given by the transition relation

$$e \xrightarrow{a} e',\ p \xrightarrow{a} p' \Rightarrow e||p \rightarrow e'||p'$$

The test aw, for example, establishes whether a process will admit action a; if so, the experimenter is happy and will report success. If a process p will always pass the test then we say that p *must* aw; if q sometimes passes the test but might sometimes *not* pass, then q *may* aw. Examples in CCS are $P \stackrel{def}{=} a.NIL$, which *must* aw; and $Q \stackrel{def}{=} a.NIL + \tau.b.NIL$, which *may* aw but not *must* aw, since Q may silently evolve to $b.NIL$ which then cannot proceed with the test. (Although testing is due to Hennessy and de Nicola [dNH84, Hen88] and is an integral part of ATP, we can nevertheless test CCS agents in a similar way.) The test $1w$ is passed by all the processes we have so far considered; the only process which fails this test under *must* is Ω, the *divergent* process whose behaviour is completely undefined; Ω *may* pass $1w$, but it is possible for Ω to evolve internally for ever, resulting in the infinite and unsuccessful computation

$$aw||\Omega \rightarrow aw||\Omega \rightarrow aw||\Omega \rightarrow \ldots$$

So the test $1w$ is needed in order to distinguish between Ω and any other process under *must* testing, since Ω *must*$\not$ $1w$. Contrast this with NIL, the process which does nothing, but whose behaviour is completely defined; we know precisely what to expect of it. We will consider the significance of Ω in more depth in 2.2.3.

Hennessy concludes that all such experiments can be reduced to one of two forms:

$$1w + b_1(1w + \ldots + b_n(1w + a)\ldots)$$

or

$$1w + b_1(1w + b_2(1w + \ldots + b_n(a_1w + \ldots + a_kw)\ldots)$$

though it is not clear that an experiment of one of these forms will make the required distinction (from the denotational and equational models) between the processes NIL and a (taking the usual liberty of omitting NIL in $a.NIL$). From the other characterizations of *must* testing, we should expect to find at least one test passed by NIL but not a and one by a but not NIL, since they are not *must* related at all. The test $1w + a$ *must* be passed by NIL but not by a since the computation

$$1w + a||a \rightarrow NIL\,||\,NIL$$

does not report success; but in the second case, where we need a test to be passed by a but not NIL, a test of the form $a_1a_2\ldots a_nw$ (in this case, simply aw), with no occurrence of $1w$, is needed in order to make the distinction. Any test containing $1w$ is always passed by NIL.

These tests are used to impose an ordering over ATP—and in the Concurrency Workbench, CCS—processes. $p \precsim_{may} q$ if every test that p *may* pass, q *may* pass also; and $p \precsim_{must} q$ if every test that p *must* pass, q *must* pass also. The testing preorder $\precsim$ between processes is merely the conjunction of these two. All three are *preorders*, that is, they are reflexive and transitive relations. Establishing preorders and the associated equivalences in the operational model is not easy, though it is the motivation for more theoretical models. Operational semantics in ATP capture our intuition about the behaviour of processes; the denotational and axiomatic models give more tractable proof systems.

2.2.3 Denotational Semantics

CCS semantics are expressed as a labelled transition system, with an axiomatization (for finite-state agents only) which is complete with respect to it. There is no clear denotational model comparable to ATP's, though Milner does represent agents as trees in a similar way[2]. In this section, though, we shall be concentrating on Hennessy's denotation.

Processes as strings

From the signature of ATP, we can interpret the terms in an algebra PS of strings, or traces. We define a mapping from NIL to the set $\{\varepsilon\}$ (in effect, the empty string), and from any process p to the union of this set with a prefix-closed subset S of Act^* (the set of strings of elements from Act), representing the sequences of actions p may perform. By *prefix-closed* we mean that if $s \in S$, then any prefix of s is also in S. So, for example, the process

$$p \stackrel{def}{=} ab + ac + b + bc$$

is interpreted as the set $\{\varepsilon, a, b, ab, ac, bc\}$; this set is also the interpretation of

$$q \stackrel{def}{=} a(b + c) + bc + NIL$$

We can think of the set as the *language* of p (and q) and refer to it as $L(p)$. The choice of this model reflects the fact that we are interested only in the sequences of actions a process *may* perform. We require the mapping from the term algebra to PS to be a homomorphism, that is, it should preserve structure. In this model, the operation $+$ in the signature is realized by set union in PS. So then, because the mapping is a homomorphism, the interpretation of the sum of two terms can be regarded as either (i) the union in PS of the interpretations of each term taken separately or (ii) the interpretation in PS of the sum of the terms in the syntax. This can be seen more clearly from the commutative diagram below, showing the interpretation of $p + q$, where $p \stackrel{def}{=} ab + ac$ and $q \stackrel{def}{=} a + bc$.

[2]This is not to say that a denotational model of CCS cannot be constructed.

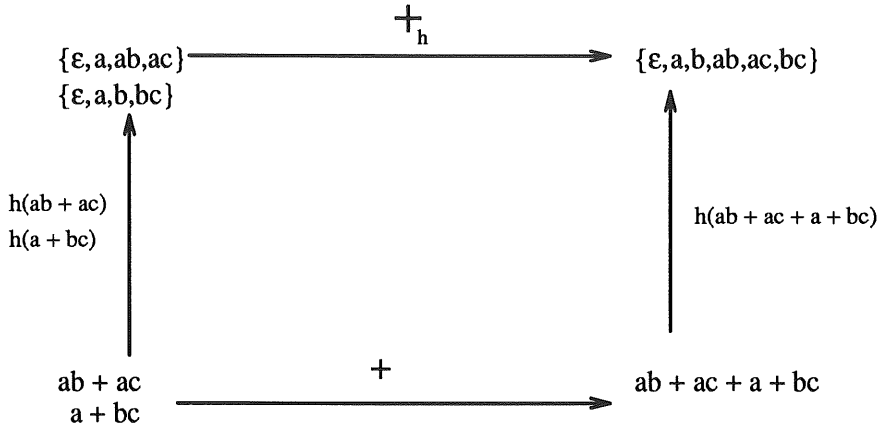

Figure 2.1: The interpretation of $p + q$

Equivalence in this model is then set equality, and the preorder (whose kernel the equivalence is) is defined by set inclusion (see chapter 4). Hennessy shows that this interpretation is equivalent to *may* testing in the operational model.

Processes as trees

We can interpret the terms of the signature as *acceptance trees*, AT. NIL is interpreted as the tree consisting of just one node and no branches. It is a *closed* node, indicating that its future behaviour is completely defined. The process Ω is also represented by a single node, but this is *open*, indicating that nothing is known about its behaviour.

Figure 2.2: The trees representing NIL and Ω

In general, processes are represented as trees whose branches are labelled by actions from *Act* and whose nodes have associated with them an *acceptance set*; the elements in this set (which are themselves sets) indicate the internal states into which the process may evolve, and the potential behaviour of the process in each state. A simple example of a finite agent is given below. The acceptance set at each node

contains only one element, indicating that the tree is fully deterministic; all choices are made by the environment. We have shown the acceptance sets at every node here, but in general, deterministic nodes are not annotated.

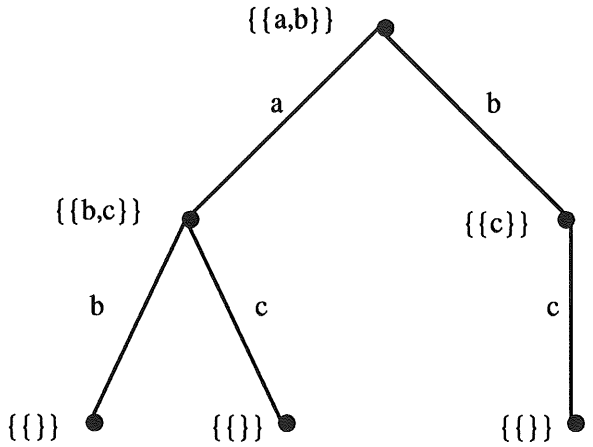

Figure 2.3: The tree $a(b+c)+bc$

In this model we are interested in processes' potential both for internal nondeterminism and for divergence; the acceptance sets reflect this. For example, the process $p \stackrel{def}{=} ab+ac$ is implicitly nondeterministic; one way to capture this in set notation is shown below:

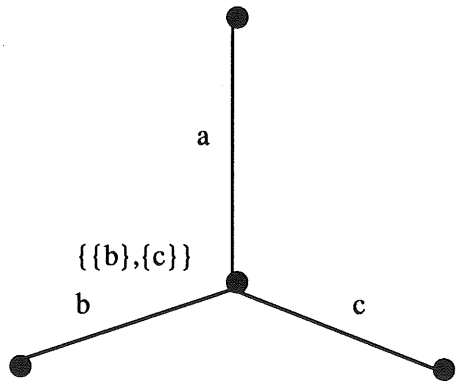

Figure 2.4: The tree $p \stackrel{def}{=} ab+ac$

Its tree is the same shape as that for $q \stackrel{def}{=} a(b+c)$, since only one branch from a node may contain any given label; the nondeterminism is captured through the acceptance

sets. The singleton sets $\{b\}$ and $\{c\}$ indicate that the process p may evolve into one of two states: in one state, only action b is possible and in other, only c. The environment does not make the choice between b and c; if it did, the acceptance set at that node would be $\{\{b,c\}\}$, as is the case for q.

This is not quite the model used by Hennessy. In ATP, trees are compared and ordered on the basis of their acceptance sets at corresponding nodes. So that processes which are equated in this model can have the same representation, acceptance sets are *saturated* or *closed*. For a given acceptance set B, this involves adding sets to B, giving $c(B)$, so that $c(B)$ is both *union closed* and *convex closed*, as follows:
(i) X, Y in B implies $X \cup Y \in c(B)$ - union closure;
(ii) $X, Y \in B, X \subseteq Z \subseteq Y$ implies $Z \in c(B)$ - convex closure.
In the example above, the closure of B, where $B \stackrel{def}{=} \{\{b\},\{c\}\}$, is given by

$$c(B) = \{\{b\},\{c\},\{b,c\}\}$$

This is also the acceptance set at a for the tree representing $p' \stackrel{def}{=} ab+ac+a(b+c)$; so we have $p = p'$. Finite processes are ordered in this model on the basis of acceptance set inclusion for corresponding nodes. (Note that nodes correspond to the strings of PS.)

We recall from PS that the choice operator $+$ was modelled as simple set union. In this model, however, things are a little less simple. Choice between trees t_1 and t_2 is modelled as the joining of trees at the roots, giving $t_3 \stackrel{def}{=} t_1 + t_2$, with t_3 determined by three rules: the language $L(t_3)$ of t_3 is given by $L(t_1) \cup L(t_2)$: the acceptance set at the root of t_3 is the pointwise union of the acceptance sets at the roots of t_1 and t_2; and the acceptance sets at the other nodes of t_3 are given by the closure of the union of the individual acceptance sets.

Infinite processes

We consider the example of a $Clock$ whose behaviour we define by

$$Clock \stackrel{def}{=} tick.Clock \quad \text{or} \quad \mathbf{fix}(X = tick.X)$$

in CCS or alternatively

$$Clock \stackrel{def}{=} rec\, x.tick\, x$$

The operational semantics for recursive terms in ATP is given by

$rec\ x.t \rightarrowtail t[rec\ x.t/x]$

that is, the behaviour of a recursive expression can be understood in terms of its successive 'unwindings'. This implies a succession of approximate meanings for the expression as the recursion is unwound, with each approximation telling us a little more about the behaviour of the whole. The approximations form a chain (see below), each of whose members is a term in the algebra and is 'better defined than' its predecessor. The limit of the chain is usually an infinite object which is taken as *the* meaning, or denotation, of the recursive expression.

Semantic domains for recursive processes need to have extra structure defined over them to reflect this. Before a recursive expression has been unwound once, we know nothing about its behaviour; we need a special element in the domain to denote this, the '$0th$' unwinding. We need to define the relation between successive approximations as the expression is unwound; and we need to accomodate the infinite object which is the limit of the chain and gives the denotation of the recursive term[3]. We take the example of the *Clock* above and look at the first few unwindings:

$0th$ unwinding	Ω
$1st$ unwinding	$tick.\Omega$
$2nd$ unwinding	$tick.tick.\Omega$
$3rd$ unwinding	$tick.tick.tick.\Omega$
...	

We see that the $0th$ unwinding is Ω, since we know nothing about the behaviour of *Clock* so far. The first unwinding tells us that the *Clock* ticks once, but then its behaviour is unknown—a little more, though, than we knew before; and so on with each unwinding. We can order these approximations to the behaviour of the *Clock* with a partial order, that is, a reflexive, transitive and antisymmetric relation, indicating our increasing understanding of the *meaning* of the *Clock*:

$$\Omega \leq tick.\Omega \leq tick.tick.\Omega \leq tick.tick.tick.\Omega \ldots$$

The sequence is called a *chain*. The partial order is the testing preorder defined over the algebra when quotiented by its kernel. Anti-symmetry follows from the fact that

[3]The denotation of a recursively defined term is not necessarily infinite; the denotation of $rec\,x.0$ is the NIL process and that of $rec\,x.x$ is Ω

we are then applying the preorder over equivalence classes. The elements in the chain are finite approximations to the infinite process; the meaning, or denotation, of the process is the (usually) infinite object which is the limit of the chain. Any function applied to the elements in the sequence is required to preserve the ordering, that is, it should be monotonic. (Interpreted, monotonicity means that we cannot get more information out of a computation than we put in.) Hennessy constructs the *ideal completion* of his algebra, which amounts to extending the algebra so as to contain all the required infinite objects. The technique depends upon the idea of *directed sets*. In a directed set, any pair of elements has an upper bound in the set, that is, an element which is 'above' both of them with respect to the partial order. (Every chain is a directed set, but not *vice versa*.) The technique of ideal completion involves adding to the algebra the least upper bound (*lub*) of each directed set, which is then the denotation of the expression of which the elements in the set are approximations[4]. In the case of recursive expressions, these *lubs* will typically be infinite objects. In this case we need to extend the notion of monotonicity to *continuity*, that is we require that any function defined over terms that are ordered in this way should also preserve the *lub*. This constraint subsumes monotonicity; it can be interpreted to mean that a finite amount of information put in to a computation can only result in a finite amount of information out (otherwise we should be able to solve the halting problem). To summarize, a semantic domain A for recursive processes must have the following structure:

(i) A is endowed with a partial order $\leq$:

(ii) A contains a least element Ω satisfying $\Omega \leq a$ for all $a \in A$:

(iii) every directed subset of A has a *lub*:

(iii) functions (from the signature, Σ), defined over A are continuous.

The structure is called a Σ *domain*.

We can represent the *Clock* as a sequence of acceptance trees, where the states which we 'understand' are represented by closed nodes and those we do not understand, i.e., whose future behaviour is not known, are represented by open nodes:

[4]The approximations to a recursive process will always form a chain. Directed sets are needed because the technique of ideal completion depends upon the fact that the union of directed sets is also a directed set; the union of chains is not in general a chain. The technique gives us a mildly generalized version of what we need for interpreting recursive processes

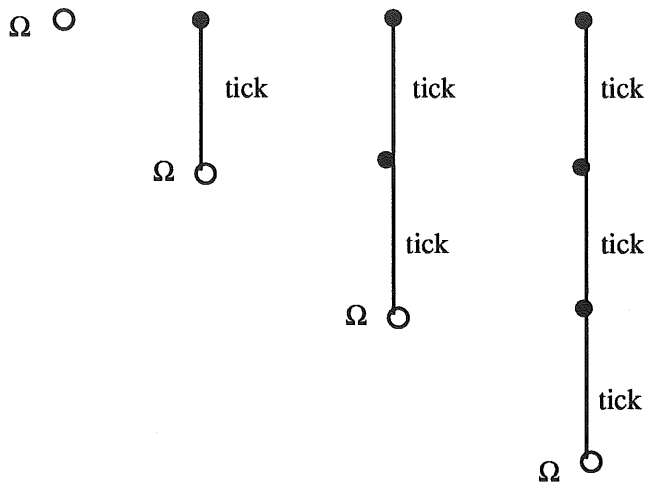

Figure 2.5: The trees for the *Clock* unwinding

The denotation of the *Clock* is an infinite, though finite branching, tree, all of whose nodes are closed.

Strong acceptance trees, AT_s, have open nodes only as leaves; this amounts to saying that if a process can diverge, we are no longer interested in its behaviour from that point (as we can see from the definition below). This models *must* testing, which is intolerant of divergence. *Weak* acceptance trees, AT_w, which model *may* testing, have every node open, reflecting an indifference to divergence. A compromise between these two, AT, imposes the condition that every descendant of an open node is also open. This models the conjunction of *may* and *must* testing, allowing us to recognize an open node and disregard it for *must* testing, but to take account of it for *may* testing.

Ordering of infinite processes

The *must* ordering of infinite processes in ATP is a generalization of the finite case which depends upon a process's capacity for divergence. Hennessy introduces a notation for this; he writes

$p\uparrow$

if p has an infinite internal computation, and its complement

$p\downarrow$

if p has no infinite internal computation. He also defines the notation

$p\downarrow s$

to mean that after the sequence s, p will always evolve to a convergent state (though this does not necessarily imply that p can actually perform the sequence s). An inclusion relation $\subset\subset$ on acceptance sets is invoked, defined as follows:

Definition 2.1: $\mathcal{A}$ and $\mathcal{B}$ satisfy $\mathcal{A} \subset\subset \mathcal{B}$ iff for every $A \in \mathcal{A}$, there exists $B \in \mathcal{B}$ such that $B \subseteq A$.

(This relation coincides with subset inclusion on the closed acceptance sets of the trees model AT.) The *may* preorder is still based upon language inclusion, but the *must* ordering is generalized as follows:

Definition 2.2: $p \sqsubseteq_{must} q$ if $p\downarrow s$ implies

(a) $q\downarrow s$ and

(b) $\mathcal{A}(q,s) \subset\subset \mathcal{A}(p,s)$

We can see from Definition 2.2 that since $\Omega\uparrow$, $\Omega \sqsubseteq_{must} p$ for all processes p; and any process with an initial divergent capability is *must* equated to Ω, whatever its *may* properties. This is discussed further in 4.3.1. In the case of the *Clock* above, we can see that each approximation as the recursion is unwound is related by $\sqsubseteq_{must}$ to its successor. It is also interesting to note that by this definition, the ordering

$$rec\, x.(ax \oplus 0) \sqsubseteq_{must} a0$$

holds; despite the small capability of the finite process on the right, which can only perform a and then dies, it is nevertheless above the infinite process on the left; this may die at any time despite the possibility of its performing an infinite sequence of a's.

2.2.4 Axiomatic Semantics

Milner gives an axiomatization of finite-state agents which is sound and complete with respect to observation congruence. Finite-state agents are those constructed using only the dynamic combinators (prefix, summation and constants), and where only finite summation and finitely many defining equations are permitted [Mil89, Mil86]. The axioms (see Appendix A) consist of some rules for recursion, the τ laws, and the monoid laws, namely, associativity, commutativity, idempotence and the identity of NIL (or 0) under summation. These laws are useful, in conjunction with the expansion law, for transforming CCS agents, though in general observation congruence is established by bisimulation construction and a demonstration of stability. The monoid laws, in particular, are strongly intuitive and tend to be assumed when transforming agent behaviours generally. The expansion law provides a method of eliminating $|$, the composition operator. It enables any composition of agents to be expressed as a sum of agents prefixed by the actions which they may immediately perform. Its use is illustrated in chapter 3. There is no finite axiomatization for CCS congruences in general [Mol89].

ATP, on the other hand, has a sound and complete axiomatization for testing equivalence. The monoid laws are as for CCS; however, the laws for internal and external nondeterminism are illuminating and draw attention to the difference from CCS. ATP has a distributive law for external choice over internal, and similarly for internal choice over external:

$$x + (y \oplus z) = (x + y) \oplus (x + z)$$
$$x \oplus (y + z) = (x \oplus y) + (x \oplus z)$$

We can illustrate the use of this by looking again at the example from 2.2.2, where we considered the processes $P \stackrel{def}{=} a + \tau.b$ and $p \stackrel{def}{=} b + (a \oplus b)$, which seemed to capture the same behaviour. Using the first distributive law together with idempotence, we can transform p as follows:

$$\begin{aligned} p &\stackrel{def}{=} b + (a \oplus b) \\ &= (b + a) \oplus (b + b) \\ &= (b + a) \oplus b \end{aligned}$$

We can see that our informal notion of the behaviour of p is still the same, even though

the internal and external operators seem to have 'changed places' (this, of course, is not generally the case; here it is because $b + b = b$); p can always perform b but might not always be able to perform a.

We can axiomatize an aspect of the trace model PS with the axiom schema
$a(x + y) = ax + ay$
generating an axiom for each $a \in Act$. This distributive law for prefixing over external choice holds when we are interested only in the sequences of actions a process may perform. It may be deduced from the axioms

$$x \oplus y \leq x + y \quad (+ \oplus 1)$$
$$x \leq x \oplus y \qquad (W)$$

using the substitution rule. (The annotation is from Hennessy's equations, page 157 of [Hen88].) The distributive law for prefixing over *internal* choice is axiomatic in both the strong and weak models.

2.3 Refinement

Refinement in software development is the process of moving from the abstract (specification) towards the concrete (implementation), usually in discrete steps and with some method of establishing that the result of each step of the development 'satisfies' its specification, namely, the result of the previous step. Each specification is in some sense 'more defined' than its predecessor, usually meaning data refinement [Jon80]. Morgan's Refinement Calculus [Mor90] provides numerous rules for the refinement of a specification; when one of the rules is applied to a specification $S1$ to give a specification $S2$, then $S1 \sqsubseteq S2$, or $S2$ *refines* $S1$. The result is a chain of specifications with the most abstract at the bottom[5]of the chain and the implementation at the top. There may be many such chains starting from the same specification, or in other words, many correct implementations of a specification. The intuition of this application of preorders seems to make good sense and is echoed in Cleaveland, Parrow and Steffen's report [CPS89] on the concurrency workbench. They write

[5]'Bottom' here refers to the starting point of the chain, i.e., the specification—not to be confused with $\perp$

> We can interpret $P \sqsubseteq Q$ as "Q is closer to an implementation than P". This interpretation is based upon the idea of regarding divergence as a means of marking underspecified states. For example, the totally divergent state $\perp$ can be seen as the totally unspecified state, allowing each process as a correct implementation. "Loose" or "partial" specifications can be used to establish that processes, although not being equivalent, may be used interchangeably in certain contexts.

This point is well taken in cases where the difference between two processes is only in divergent behaviour. However, as we see in the next chapter, when a specification is written in CCS and 'refined' by successive decomposition of agents, then the preorder relation holds in the other direction. Hennessy, in the introduction to his book 'Algebraic Theory of Processes' [Hen88] says of the testing preorder (over finite processes)

> ...$p \precsim q$ implies that q is a more deterministic version of p. So $\precsim$ could be taken as a formalization of "is a correct implementation of".

contrasting with the quotation above from [CPS89]. This is illustrated and discussed further in the next chapter; it is worth noting, however, that for finite processes, $\precsim$ implies $\approx_{may}$. This implication does not hold for recursive processes; $\approx_{may}$ must be established explicitly.

Another kind of refinement, called *action refinement*, is proposed by Aceto and Hennessy [AH90, AH88]. Here, an action at one level of abstraction is refined to a process at a lower level, while still preserving the synchronization structure of processes. Lamport [Lam83, AL88] proposes *refinement mappings* between specifications, the mapping being between the state spaces (both external and internal) of the specifications, from the steps of one state machine to the steps of the other. None of these methods has been considered in depth, but the fact of their development suggests a clear recognition of the need for a practical application of mathematical and logical theories.

Chapter 3

A CCS case study

In this chapter we specify a safety-critical system, namely a level crossing, in CCS. We aim to show that the specification, including safety requirements, may be captured entirely within the language. In this way the analytical power of CCS can be used to give a full and unified account of the system's behaviour. The informal description of the level crossing is taken from [Gor87]. In that paper the system requirements, which are mainly safety-related, are specified in temporal logic; here we show that, with a few reservations, the sequential and synchronization features of CCS enable us to express the necessary causality within the language itself.

We begin with an informal description of the level crossing system as set out in [Gor87]. In section 2 we give the main safety requirements, also taken from [Gor87] and show how these may be translated into CCS. Section 3 consists of the CCS specification, including a modified version in which the single system is divided into two communicating subsystems, this being more manageable than the whole. In section 4 we show that the two versions are congruent and having proved this, we then examine the behaviour of the composed subsystems in section 5. Lastly, we show that the system as specified satisfies the safety requirements, though there is no bisimulation between the specification and the design.

3.1 Informal description of the system

Fig.1 shows the layout of the crossing; the names and arrangement of the physical objects are taken from [Gor87]. The two sensors on the approach side of the line are located after the appropriate lights (in the direction of motion). The required behaviour of the train driver is that he stops the train on a red $A(pproach)Light$ and proceeds when the light is green; in proceeding, the train necessarily crosses the sensor TA in the line. A signal is sent by TA to $Control$ which then changes the $I(n)Light$ to green as soon as the crossing has been cleared of cars. When the train crosses the sensor TI, a signal is sent to $Control$ which then changes the $ILight$ back to red. Only after the train is clear of the crossing, that is, it has crossed TO, is the gate reopened and the $RLight$ changed back to green. Cars making legitimate use of the crossing are sensed by $RSensor$ and are counted in and out. Required behaviour of drivers is that they stop their vehicles when the light is red and do not proceed until the light changes to green. This is not specified explicitly. Drivers attempting to use the crossing after the light has changed to red are taking calculated risks and should be aware that no account of this is taken in specifying the safety features of the system.

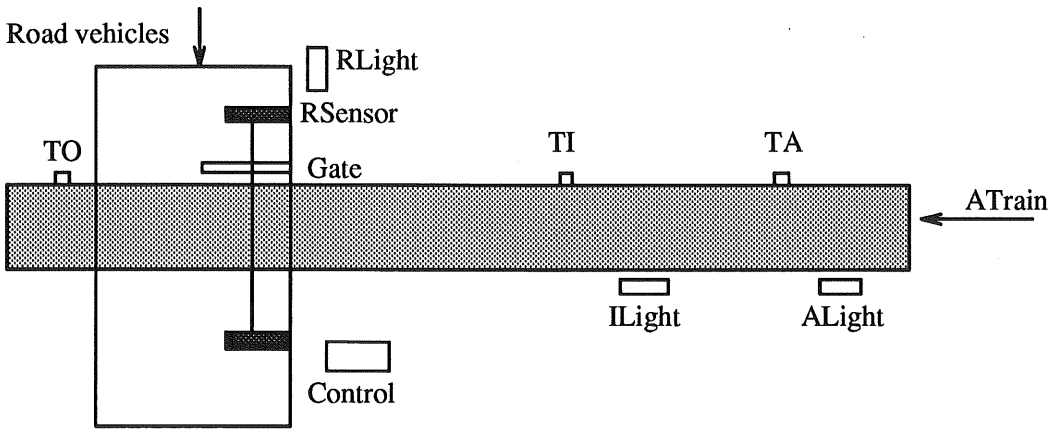

Figure 3.1: The crossing layout

3.2 Safety requirements

Global model

The top level safety requirement $S1$, or 'global model' [Gor87], is that there should never be a train *and* a car inside the crossing at the same time. This main safety requirement is achieved by the following lower level constraints on the application domain. The temporal logic translations of $S1$ and the conditions below can be found in [Gor87].

Application domain constraints

(1) For every train t, if t is outside the crossing and the railway light is red, then t remains outside unless it 'sees' the green light.
(2) After the road light has been switched to red, cars in the crossing will be allowed to leave before the barrier is lowered.
(3) If the crossing is open for cars then the rail light must be red and there must be no train in the crossing.
(4) If the crossing is open for trains then the road is blocked (gate down) before a train enters the crossing.
(5) If the gates are closed then the road lights must be red.

3.3 The CCS specification

The global model, $S1$, may be translated into CCS:

$$
\begin{aligned}
SafeCrossing \quad &\stackrel{def}{=} \quad \overline{train_{in}}.\overline{train_{out}}.SafeCrossing \qquad (S1_{ccs}) \\
&\qquad\quad + \\
&\qquad\quad \overline{car_{in}}.SafeCrossing_{cars}(1) \\
SafeCrossing_{cars}(n) \quad &\stackrel{def}{=} \quad \overline{car_{in}}.SafeCrossing_{cars}(n+1) \\
&\qquad\quad + \\
&\qquad\quad \overline{car_{out}}.\ \text{if } n = 1 \text{ then } SafeCrossing \\
&\qquad\quad\ \ \text{else } SafeCrossing_{cars}(n-1)
\end{aligned}
$$

SafeCrossing is regarded as a shared resource; it may be used either by cars or by trains but not by both together (since *car* and *train* signals cannot be interleaved). It does not deadlock; there is no state in which both cars and trains are prevented from using the crossing. Cars are counted in and out of the crossing and only when it is clear of cars does the system allow trains to enter.

3.4 The CCS design structure

Sensors and track lights

The sensors, $TA(pproach)$, $TI(n)$ and $TO(ut)$, are triggered by the train (t_a, t_i, t_o) and send signals to the control (a, i, o) to indicate the train's position (see Figure 3.1). The track lights are parameterized by colour; the *ALight* is initially green and the *ILight* initially red. These change on receiving the signal from *Control*. A function *change* is assumed with domain $\{red, green\}$, where $change(red) = green$ and $change(green) = red$.

$$\mathbf{TA} \stackrel{def}{=} t_a.\overline{a}.\mathbf{TA}$$
$$\mathbf{TI} \stackrel{def}{=} t_i.\overline{i}.\mathbf{TI}$$
$$\mathbf{TO} \stackrel{def}{=} t_o.\overline{o}.\mathbf{TO}$$
$$\mathbf{ALight}(x) \stackrel{def}{=} \overline{send_a}(x).change_a.\mathbf{ALight}(change(x))$$
$$\mathbf{ILight}(y) \stackrel{def}{=} \overline{send_i}(y).change_i.\mathbf{ILight}(change(y))$$

The train

Condition (1) of 3.2 is made explicit in the specification of the train. An approaching train sees the *ALight* and stops if it is red. It 'polls' this light until the signal changes to green, when it can proceed and cross TA. Its behaviour is similar at the *ILight*. When this is green and the train has crossed TI it sends an observable signal $train_{in}$ to the environment. When the train leaves the crossing it sends an observable signal $train_{out}$ just before crossing TO.

$$\mathbf{ATrain} \stackrel{def}{=} send_a(red).\mathbf{ATrain} + send_a(green).\overline{t_a}.\mathbf{ITrain}[1]$$

$$\mathbf{ITrain} \stackrel{def}{=} send_i(red).\mathbf{ITrain} + send_i(green).\overline{t_i}.\mathbf{CTrain}$$

$$\mathbf{CTrain} \stackrel{def}{=} \overline{train_{in}}.\overline{train_{out}}.\overline{t_o}.\mathbf{ATrain}$$

Road sensor

The Road sensor works in two ways. Firstly, it responds to the road light; when this is red, the sensor simply waits for it to turn to green. On the green signal, waiting cars are admitted to the crossing and are counted in and out by *Cars*. If there are no cars waiting the sensor will just wait for the lights to change back to red, or for a car to arrive, whichever is first. Each car is observed entering and leaving the crossing through car_{in} and car_{out} signals. Only when all cars have left, that is, when the number of *out* signals is equal to the number *in*, does the sensor check the lights again, so satisfying condition (2) in 3.2.

The lights might have changed to red in the meantime, modelling the situation in which cars may choose to ignore red lights and hence prevent the train from using the crossing. The sequential nature of *Control* makes it safe for them to do this. It is important to note here, however, that while the sensor is reading the red light it is not sensing cars on the crossing. Having checked once that the crossing is clear it sends the acknowledgement *sent* to *Control* which then begins to close the gate. If at this point a car enters the crossing it will not be sensed and an accident may be caused; responsible behaviour of drivers is assumed (see caveat in 3.1). A car using the crossing legitimately and breaking down before it has left will be sensed and will therefore be safe.

[1]Strictly, $send_a$ should be parameterized and followed by an 'if then else' statement

$$
\begin{aligned}
\textbf{RSensor} \stackrel{def}{=}\ & send_r(red).\overline{sent}.\textbf{Stop} \\
& + \\
& send_r(green).\textbf{Go}
\end{aligned}
$$

$$
\begin{aligned}
\textbf{Stop} \stackrel{def}{=}\ & send_r(green).\overline{sent}.\textbf{RSensor} \\
\textbf{Go} \stackrel{def}{=}\ & \overline{car_{in}}.\textbf{Cars}(1) + send_r(red).\overline{sent}.\textbf{Stop} \\
\textbf{Cars}(n) \stackrel{def}{=}\ & \overline{car_{in}}.\textbf{Cars}(n+1) \\
& + \\
& \overline{car_{out}}.\ \text{if } n = 1 \text{ then } \textbf{RSensor} \\
& \text{else } \textbf{Cars}(n-1)
\end{aligned}
$$

Road lights and gate

RLight (which is initially green) is read by the road sensor. It continues to send the 'red' signal until it is told by *Control* to change. The gate receives the signal (a toggle) to change its state and having done so, sends the acknowledgement 'done' back to *Control*.

$$
\begin{aligned}
\textbf{RLight}(z) \stackrel{def}{=}\ & \overline{send_r(z)}.\textbf{RLight}(z) \\
& + \\
& change_r.\overline{send_r}(change(z)).\textbf{RLight}(change(z))
\end{aligned}
$$

$$
\textbf{Gate} \stackrel{def}{=} \quad movegate.\overline{done}.\textbf{Gate}
$$

Main control

All communications in conditions (3) to (5) in 3.2 take place through *Control*. These are entirely sequential, so it can be seen by inspection in what order the operations are performed, thus satisfying the conditions. *Control*, having sensed an approaching train, first changes the *ALight* (furthest up the track) to red to prevent any more trains entering the section. It then changes the road light to red and on receiving an acknowledgement, closes the gate. Only when this has been acknowledged does it change the *ILight* (on the track, closest to the crossing) to green to allow the train to

proceed through the crossing. When it senses the train inside the crossing the *ILight* is changed back to red. When the train has left the crossing (both track lights are still red) the gate is opened, the road lights are changed to green and the traffic allowed to proceed. Lastly, the *ALight* is changed back to green.

$$\begin{aligned}\textbf{Control} \stackrel{def}{=}\ & a.\overline{change_a}.\overline{change_r}.sent.\overline{movegate}.done.\\ & \overline{change_i}.i.\overline{change_i}.o.\\ & \overline{movegate}.done.\overline{change_r}.sent.\overline{change_a}.\textbf{Control}\end{aligned}$$

The composed system

The composed crossing system is given by

$$\begin{aligned}\textbf{Crossing} \stackrel{def}{=}\ & \textbf{ATrain}|\textbf{TA}|\textbf{TI}|\textbf{TO}|\textbf{ALight}(green)|\textbf{ILight}(red)\\ & |\textbf{Control}|\textbf{RLight}(green)|\textbf{Gate}|\textbf{RSensor} \setminus A\end{aligned}$$

where

$$\begin{aligned}A = \ & \{a, i, o, t_a, t_i, t_o, change_a, change_i, change_r,\\ & send_a, send_i, send_r, sent, movegate, done\}\end{aligned}$$

(see Figure 3.2) so that the only actions visible to the environment are $train_{in}$, $train_{out}$, car_{in}, car_{out}.

We now wish to analyse the behaviour of the Crossing as a whole. We can do this by applying the expansion law of CCS to the composed system; this provides a mechanism whereby parallel processes may be unwound into purely nondeterministic ones. Informally, it expresses the actions of a parallel (that is, interleaved) composition of processes in terms of the the actions of the individual processes themselves. Chapter 3 of [Mil89] gives a clear and readable account of this law. However, attempts to apply the theorem show that possible states for the whole system proliferate within one or two lines; the proof becomes unmanageably complex. We should like, if possible, to decompose the total system into smaller more manageable subsystems which may be analysed independently and then composed. The following is an attempt to achieve this.

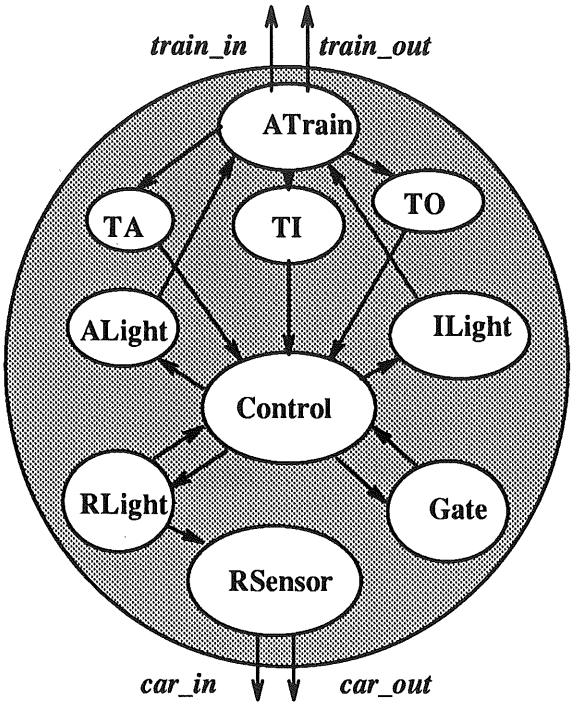

Figure 3.2: The composed system 'Crossing'

3.5 Divide and conquer

We propose to divide the crossing system into two subsystems: the road vehicles with *R_Control* and the trains with *T_Control*. Only *Control* needs to be changed to do this; it is split into two smaller but communicating control subsystems, one each for the road vehicles and the trains. The extra signals introduced are *stopcars*, *gotrains*, *gocars* and *restore*; these are used to synchronize the two subsystems and prevent any interleaving of trains and cars.

Road control

R_Control cannot act until it has received the *stopcars* signal from *T_Control*. On receiving this, *R_Control* changes the road lights to red and on receiving an acknowledgement closes the gate. When this has been acknowledged, the signal *gotrains* is sent to *T_Control* and the signal *gocars* awaited. On receiving this, *R_Control* then changes the road lights back to green and opens the gate. Finally the *restore* signal is sent to *T_Control*.

$$\begin{aligned}\textbf{R\_Control} \stackrel{def}{=}\ & stopcars.\overline{change_r}.sent.\overline{movegate}.done. \\ & \overline{gotrains}.gocars.\overline{movegate}.done.\overline{change_r}.sent. \\ & \overline{restore}.\textbf{R\_Control}\end{aligned}$$

Train control

T_Control having sensed an approaching train first changes the *ALight* to red to prevent any more trains entering the section. It then sends the signal *stopcars* to *R_Control* and cannot proceed until it has received the signal *gotrains*. On receiving this from *R_Control*, it sends the signal to change the *ILight* to green to allow the train to proceed through the crossing. When it senses the train inside the crossing the *ILight* is changed back to red. When the train has left the crossing (both track lights are still red) the signal *gocars* is sent to *R_Control* and the signal *restore* is awaited. On receiving this the *ALight* is changed back to green.

$$\begin{aligned}\textbf{T\_Control} \stackrel{def}{=}\ & a.\overline{change_a}.\overline{stopcars}.gotrains. \\ & \overline{change_i}.i.\overline{change_i}.o.\overline{gocars}.restore. \\ & \overline{change_a}.\textbf{T\_Control}\end{aligned}$$

The composed road vehicle and *R_Control* subsystem

The observable communications for the road subsystem are car_{in}, car_{out}, *stopcars*, *gotrains*, *gocars* and *restore*.

$$\begin{aligned}\textbf{RX} = \ & \textbf{RLight}(green)|\textbf{Gate}|\textbf{RSensor}|\textbf{R\_Control} \\ & \backslash\{change_r, movegate, send_r, sent, done\}\end{aligned}$$

The composed train and *T_Control* subsystem

The observable communications for the train subsystem are $train_{in}$, $train_{out}$, *stopcars*, *gotrains*, *gocars* and *restore*.

$$\begin{aligned}\textbf{TX} = \ & \textbf{TA}|\textbf{TI}|\textbf{TO}|\textbf{ALight}(green)|\textbf{ILight}(red)|\textbf{T\_Control}|\textbf{ATrain} \\ & \backslash\{a, i, o, t_a, t_i, t_o, change_a, change_i, send_a, send_i\}\end{aligned}$$

The subsystems composed

$$\mathbf{Crossing2} = ((\mathbf{TX} \setminus T)|(\mathbf{RX} \setminus R)) \setminus C$$

where

$$\begin{aligned} T = &\ \{a, i, o, t_a, t_i, t_o, change_a, change_i, send_a, send_i\}, \\ R = &\ \{change_r, movegate, send_r, sent, done\} \text{ and} \\ C = &\ \{stopcars, gotrains, gocars, restore\} \end{aligned}$$

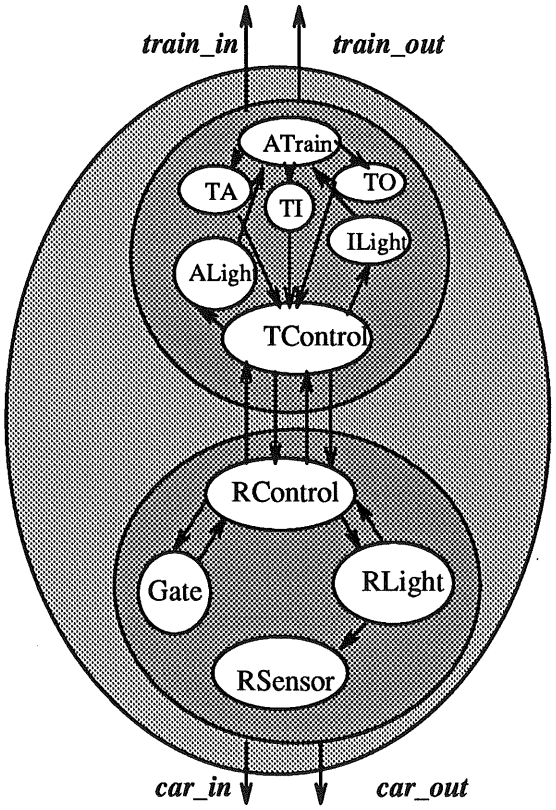

Figure 3.3: The decomposed system 'Crossing2'

Before proceeding, we need to show that that **Crossing2** above and **Crossing** (page 36) are congruent. We can then consider the composed subsystems in the confidence that whatever is shown to be true for **Crossing2** will also be true for **Crossing**.

Equivalence of the single and decoupled systems

Before continuing, we recall the particular Restriction Laws which we shall be invoking, and define some notation.

The Restriction Laws [Mil89]

(1) $P \setminus L = P$ if $\mathcal{L}(P) \cap (L \cup \overline{L}) = \emptyset$

(2) $P \setminus K \setminus L = P \setminus (K \cup L)$

(3) ...(not invoked)

(4) $(P|Q) \setminus L = (P \setminus L)|(Q \setminus L)$ if $\mathcal{L}(P) \cap \overline{\mathcal{L}(Q)} \cap (L \cup \overline{L}) = \emptyset$

Notation

T, R and C are as defined in 3.5. $\mathcal{L}(P)$ is the *sort* of P, that is, its label set. An overbar above the name of such a set refers to the complement of that set. So

$$\mathcal{L}(TX) = \{a, i, o, t_a, t_i, t_o, change_a, change_i, send_a, send_i, \overline{stopcars}, gotrains, \overline{gocars}, restore, train_{in}, train_{out}\}$$

$$\mathcal{L}(RX) = \{change_r, movegate, send_r, sent, done, stopcars, \overline{gotrains}, gocars, \overline{restore}, car_{in}, car_{out}\}$$

From this we note that

(i) $T \cup R = A$ (as defined in 3.4)

(ii) $\mathcal{L}(TX) \cap (R \cup \overline{R}) = \emptyset$

(iii) $\mathcal{L}(RX) \cap (T \cup \overline{T}) = \emptyset$

Proving equivalence

First we need the following lemma:

Lemma 3.1 $Control = (T\_Control | R\_Control) \setminus C$

The proofs of this lemma and Theorem 3.1 below are given as an indication of the complexity of the technique. The proofs of all following lemmas and theorems are given in Appendix C.

Proof

$$
\begin{aligned}
(\mathbf{T\_Control}|\mathbf{R\_Control}) \setminus C \quad &= (a.\overline{change_a}.\overline{stopcars}.gotrains.\overline{change_i}.i. \\
&\quad \overline{change_i}.o.\overline{gocars}.restore.\overline{change_a}. \\
&\quad \mathbf{T\_Control}|stopcars.\overline{change_r}.sent. \\
&\quad \overline{movegate}.done.\overline{gotrains}.gocars. \\
&\quad \overline{movegate}.done.\overline{change_r}.sent.\overline{restore}. \\
&\quad \mathbf{R\_Control}) \setminus C \\
&= (a.\overline{change_a}.\tau.\overline{change_r}.sent. \\
&\quad \overline{movegate}.done.\tau.\overline{change_i}.i.\overline{change_i}.o. \\
&\quad \tau.\overline{movegate}.done.\overline{change_r}.sent.\tau.\overline{change_a}. \\
&\quad (\mathbf{T\_Control}|\mathbf{R\_Control}) \setminus C) \\
&= (a.\overline{change_a}.\overline{change_r}.sent. \\
&\quad \overline{movegate}.done.\overline{change_i}.i.\overline{change_i}.o. \\
&\quad \overline{movegate}.done.\overline{change_r}.sent.\overline{change_a}. \\
&\quad (\mathbf{T\_Control}|\mathbf{R\_Control}) \setminus C) \\
&= \mathbf{Control} \ \square
\end{aligned}
$$

We use this congruence in the following theorem.

Theorem 3.1 $Crossing2 = Crossing$

Proof

$$
\begin{aligned}
\textbf{Crossing2} &= ((TX \setminus T)|(RX \setminus R)) \setminus C \\
&= ((TX \setminus T \setminus R)|(RX \setminus R)) \setminus C \\
&\quad \text{(by (1) and (ii))} \\
&= (TX \setminus T|RX) \setminus R \setminus C \\
&\quad \text{(by (4))} \\
&= (TX \setminus T|RX \setminus T) \setminus R \setminus C \\
&\quad \text{(by (1) and (iii))} \\
&= (TX|RX) \setminus T \setminus R \setminus C \\
&\quad \text{(by (4))} \\
&= (TX|RX) \setminus C \setminus T \cup R \\
&\quad \text{(by (2))} \\
&= (ATrain|TA|\ldots|T\_Control)|(R\_Control|\ldots|RSensor) \\
&\quad \setminus C \setminus T \cup R) \text{ (by commutativity of '|')} \\
&= ATrain|TA|\ldots|(T\_Control|R\_Control)|\ldots|RSensor \\
&\quad \setminus C \setminus T \cup R \text{ (by associativity)} \\
&= (ATrain \setminus C|TA \setminus C|\ldots|(T\_Control|R\_Control) \setminus C| \\
&\quad \ldots|RSensor \setminus C) \setminus T \cup R \text{ (by (4))} \\
&= (ATrain|TA|\ldots|(T\_Control|R\_Control) \setminus C| \\
&\quad \ldots|RSensor) \setminus T \cup R \text{ (by (1))} \\
&= (ATrain|TA|\ldots|Control|\ldots|RSensor) \setminus T \cup R \\
&\quad \text{(by lemma 3.1)} \\
&= (ATrain|TA|\ldots|Control|\ldots|RSensor) \setminus A \\
&\quad \text{by (i)} \\
&= \textbf{Crossing} \qquad \square
\end{aligned}
$$

3.5.1 Analysis of the subsystems' behaviour

We use the theorem to analyse the behaviour of the two subsytems.

Expansion of the train subsystem

The following result gives the behaviour of the train subsystem as observed by its environment at the lower level of composition:

Lemma 3.2

$$\begin{aligned} TX \setminus T = \; & \tau.\overline{stopcars}.gotrains.\overline{train_{in}}. \\ & \overline{train_{out}}.\overline{gocars}.restore.TX \setminus T \end{aligned}$$

Expansion of the road subsystem

The following gives the behaviour of the road subsytem:

Lemma 3.3

$$\begin{aligned} RX \setminus R \; = \; & \tau. \\ & (\overline{car_{in}}.(\overline{car_{in}}.\overline{car_{out}})^*.stopcars.(\overline{car_{in}}.\overline{car_{out}})^*.\overline{car_{out}}. \\ & \overline{gotrains}.gocars.\overline{restore}.RX \setminus R \\ & + \\ & stopcars.\overline{gotrains}.gocars.\overline{restore}.RX \setminus R \\ & + \\ & \overline{car_{in}}.(\overline{car_{in}}.\overline{car_{out}})^*.\overline{car_{out}}.RX \setminus R) \\ & + \\ & stopcars.\tau. \\ & (\tau.\overline{car_{in}}.(\overline{car_{in}}.\overline{car_{out}})^*.\overline{car_{out}}.\overline{gotrains}.gocars.\overline{restore}.RX \setminus R \\ & + \\ & \tau.\overline{gotrains}.gocars.\overline{restore}.RX \setminus R) \end{aligned}$$

Expansion of Crossing2

From Theorem 3.1

$$\begin{aligned}\textbf{Crossing} &= \textbf{Crossing2} \\ &= ((\mathbf{RX} \setminus R)|(\mathbf{TX} \setminus T)) \setminus C\end{aligned}$$

By composing the subsystems above, we prove the following result, describing the behaviour of the composed crossing system:

Theorem 3.2

$$\begin{aligned}Crossing \;=\; & \tau.(\overline{car_{in}}.(\overline{car_{in}}.\overline{car_{out}})^*.\overline{car_{out}}.\overline{train_{in}}.\overline{train_{out}}.Crossing \\ & + \\ & \tau.\overline{train_{in}}.\overline{train_{out}}.Crossing \\ & + \\ & \overline{car_{in}}.(\overline{car_{in}}.\overline{car_{out}})^*.\overline{car_{out}}Crossing) \\ & + \\ & \tau.(\tau.\overline{car_{in}}.(\overline{car_{in}}.\overline{car_{out}})^*.\overline{car_{out}}.\overline{train_{in}}.\overline{train_{out}}.Crossing \\ & + \\ & \tau.\overline{train_{in}}.\overline{train_{out}}.Crossing)\end{aligned}$$

We can simplify the appearance of this by making the following substitution:
Let

- $\sigma_1 = \overline{car_{in}}.(\overline{car_{in}}.\overline{car_{out}})^*.\overline{car_{out}}$, (so $\#car_{in} = \#car_{out}$) .
- $\sigma_2 = \overline{train_{in}}.\overline{train_{out}}$ (either of these may include 0 or more τ's)
- $C = Crossing.$

Then

$$\begin{aligned}C = \; & \tau.(\sigma_1.\sigma_2.C + \tau.\sigma_2.C + \sigma_1.C) \\ & + \\ & \tau.(\tau.\sigma_1.\sigma_2.C + \tau.\sigma_2.C)\end{aligned}$$

(See Fig.3.4)

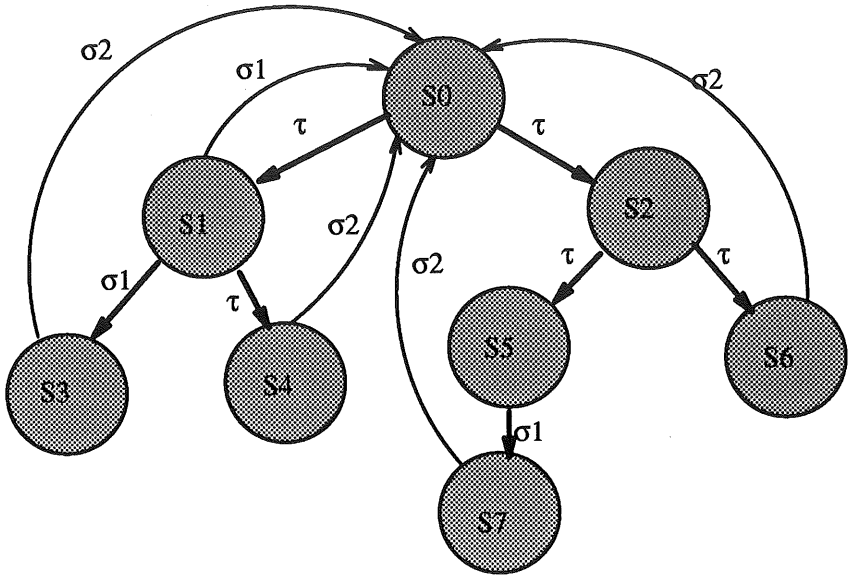

Figure 3.4: The graph of 'C'

3.6 Discussion

We recall $S1_{ccs}$ from 3.3:

$$
\begin{aligned}
SafeCrossing \quad &\stackrel{def}{=} \quad \overline{train_{in}}.\overline{train_{out}}.SafeCrossing \\
&\qquad\quad + \\
&\qquad\quad \overline{car_{in}}.SafeCrossing_{cars}(1) \\
SafeCrossing_{cars}(n) \quad &\stackrel{def}{=} \quad \overline{car_{in}}.SafeCrossing_{cars}(n+1) \\
&\qquad\quad + \\
&\qquad\quad \overline{car_{out}} \text{ if } n = 1 \text{ then } SafeCrossing \\
&\qquad\quad \text{else } SafeCrossing_{cars}(n-1)
\end{aligned}
$$

Making the substitution as above with $SC = SafeCrossing$, we have

$$SC = \sigma_2.SC + \sigma_1.SC$$

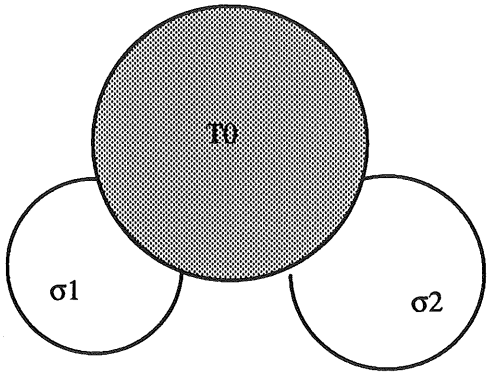

Figure 3.5: The graph of SC

Clearly this is not observation equivalent to C. Every choice in C is unstable, that is, it is (weakly) guarded by one or more τ actions, representing either the reading of a green light by $RSensor$, the sensing of an approaching train by $T\_Control$, or some other signal informing the system behaviour; the nondeterminism is *internal* to the system. In $S1_{ccs}$, however, the observer may make the choice between cars and trains; the nondeterminism is *external*. We could remove this choice from the observer and give it to the system by (weakly) guarding each summand in $SafeCrossing$ with a τ thus:

$$\begin{aligned} SafeCrossing' \stackrel{def}{=} \quad & \tau.\overline{train_{in}}.\overline{train_{out}}.SafeCrossing' \quad (S1'_{ccs}) \\ & + \\ & \tau.\overline{car_{in}}.SafeCrossing_{cars}(1) \end{aligned}$$

Now, however, we are presented with the possibility of deadlock; the system might 'slip' into the first state, allowing trains to enter (so barring cars from doing so) when no train is approaching. τ's arise because of hidden internal communications whose nature is not known to the observer; all τ's look the same, whether they arise from a whim of the system or from responsible decision making procedures; and these may include desirable failsafe mechanisms. We cannot distinguish between 'good' and 'bad' τ's, and without knowing the nature of the hidden actions we cannot guarantee liveness.

What we wish to model is the situation where the observer sees either a train or cars (but not both at the same time) and is never prevented from seeing either, but cannot choose between them; a proper 'choice' is made by the system, e.g., when an

approaching train is sensed, actions are performed which will eventually allow the train to cross. It might be possible to model this by guarding the choices in $SafeCrossing$ with visible actions but this is moving further away from the real situation; the observer, from his helicopter presumably, sees cars and trains and nothing else. The kind of nondeterminism we wish to model falls somewhere between these two extremes.

We leave the question open for the present and make the above substitution in $S1'_{ccs}$, giving

$$SC' = \tau.\sigma_2.SC' + \tau.\sigma_1.SC'$$

There is, however, no bisimulation between this and C as can be seen from the Concurrency Workbench (see Appendix D). The problem arises when trying to match the actions of SC' with corresponding actions of C. Referring to figures 3.4 and 3.6 and taking (S_0, T_0) as the starting point, T_0 may perform action τ to become T_1; this means that S_0 must either perform a τ action, so moving into S_1 or S_2 and entailing (S_1, T_1) or (S_2, T_2) as a member of $\mathcal{R}$, or remain in the same state, entailing (S_0, T_1) as a member of $\mathcal{R}$. None of these closes the set $\mathcal{R}$ and so there is no bisimulation.

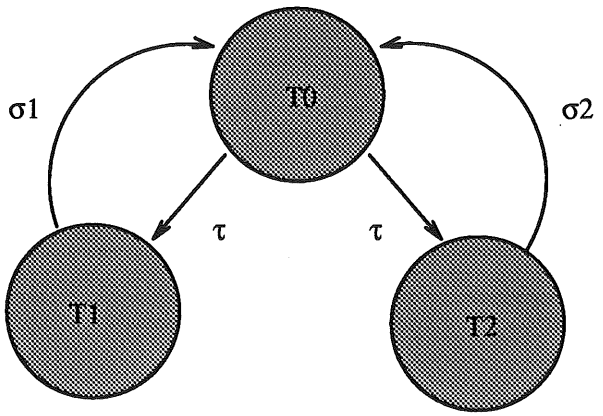

Figure 3.6: The graph of SC'

The hidden communications in the decomposition have given rise to an unstable output from the expansion law. This has precluded equivalence between the composite system and the specification for the reasons described above. In view of this unavoidable feature of the calculus, it would seem that we need to find some other satisfaction relation between specifications—something less than equality, or even equivalence, but

none the less well-defined and readily interpreted in the problem domain.

Several issues arise from this case study other than the particular one which will be addressed in later chapters (that is, the problem of establishing a satisfaction relation between CCS specifications and their refinements). Many of them could have formed the basis for a research project and it is hoped that some will indeed be the subject of further work.

The level crossing is a 'real world' application of CCS and as such, illustrates some of the strengths and limitations of the language. Despite being a small system it is nevertheless too large to be analysed as a whole due to the proliferation of states quite early on in the analysis; and the concurrency workbench cannot, at the time of writing, deal with parameter passing. (We could have rewritten the specifications in pure CCS, but would have run into problems with the potentially infinite stream of cars). However it was possible within the language to separate the single system into smaller communicating subsystems, provably equivalent to the original, with each subsystem smaller and so more amenable to analysis (though even here the states are numerous).
The behaviour of the system may be summarized as follows:

- it has been shown that there can be no interleaving of cars and trains and so the main safety requirement is satisfied;
- apparently the system can deadlock, since we know nothing about the nature of the τ's; the system may 'choose' to stay indefinitely in, say, the state which allows cars to use the crossing and so prevent trains from doing so. We need to assume that the hidden actions are taken responsibly and safely by the system, but there is no mechanism within CCS for making this explicit and so liveness cannot be guaranteed. Without the τ's, $SafeCrossing$ seems to allow the observer to experiment with the system, rather than merely to observe it and this also is undesirable. Nevertheless, it can still be seen that at no time are both cars and trains prevented from using the crossing:
- the system may also livelock; a steady stream of cars all ignoring the lights may continue indefinitely, blocking the trains; there is no mechanism in CCS for stat-

ing that a communication *must* happen eventually. Cleaveland and Hennessy's [CH88] prioritizing of actions in CCS might well solve this:

- the system fails safe; cars must be counted out of the crossing before a train may enter and the train must leave before cars may re-enter. If either a train or a car enters the crossing but fails to leave (whether it breaks down or for any other reason), the appropriate lights will remain on red, preventing other vehicles from entering the crossing:
- the required behaviour of the train (i.e., the driver) has been specified explicitly. However, in the case of road vehicles, responsible behaviour of drivers has been assumed. Drivers who attempt to use the crossing unlawfully do so at their own risk.

By definition, CCS cannot detect the absence of a signal, only its presence. Since all communications are synchronized, it is not possible, for instance, for *Rlight* just to 'show' green; something must be there to 'see' it and so allowance has to be made for this in the specification of *RSensor*. $If \ldots then \ldots else$ statements may be used to test the parameters of an action, but not to test for the presence of the action itself; we cannot model interrupts in CCS. This is not insurmountable here but in other circumstances might give problems.

We were able to model the temporal safety constraints set out in [Gor87] in two ways. First, it *is* possible to express temporal ordering in CCS and it is this ordering, rather than time itself, which was required in safety conditions (3) to (5) of 3.2. Second, the fact that all signals produced have to be consumed means that an *acknowledgement* signal cannot be lost; *Control* will wait indefinitely before allowing further actions and so the system as specified fails safe. On the other hand, as is stated in the proof of Lemma 3.2, it is possible in the CCS system for the visible actions car_{in}, car_{out} to prevent the $change_r$ action from being sent by *R_Control* to *RLight*; interpreted, this means that a steady stream of cars may prevent the train from using the crossing indefinitely; fairness in this case is not guaranteed. There is no way of forcing this communication in CCS, whereas in temporal logic it could be made explicit that this action must happen eventually (though as mentioned above, the prioritizing of actions [CH88] might be the answer). Process Logic (Hennessy–Milner[HM85]) presented in

chapter 10 of [Mil89] is a modal logic which can be used as a language in which to specify system requirements and give a characterization of observation equivalence. In this system we could specify that a particular action *must* eventually occur; in this case, though, we would be specifying a different crossing from the above. Any system which is a direct mapping from CCS to PL will have the same shortcomings.

Condition (2), (which said that cars must have time to leave the crossing before the gate is lowered), seems to need some means of measuring time explicitly (and this could be expressed in Timed CCS [MT89, Tof89]). However, in the specification of $RSensor$ we have achieved the same end but by different means. Instead of measuring time, we count cars in and out and only allow the gate to be closed after the crossing has cleared. This is inherently safer than allowing a fixed period of time to elapse.

Several problem areas have been identified; unfortunately, time does not permit us to examine them all in detail. The particular problem which we have chosen to address in the following chapters is that of establishing a satisfaction relation between CCS specifications and their refinements. Though the worked examples given in [Mil89] (such as the $jobshop$) result in stable agents which are then provably equivalent to their specifications, in general a stable output from the expansion law is by no means guaranteed, especially in the case of safety-critical systems where signals may need to be sent and received *before* a visible action is allowed to occur.

Before examining possible solutions to this problem, then, we need to take a closer look at the nature of certain notions of equivalence; this is the subject of the next chapter.

Chapter 4

Equivalence

In this chapter we consider some notions of equivalence, beginning with an *operational* view and the intuition we are trying to model. In section 2 we look at strong and weak *bisimulation* with some examples to illustrate the advantages and limitations of these as models of equivalence. We make a brief excursion into a version of *modal logic* as it is interpreted for CCS processes and see how it can be used to give an alternative characterization of *observation equivalence.* We look at two *orderings* over CCS agents, namely *simulation* and *prebisimulation*, the latter forming the basis for the implementation of the testing preorders over CCS agents in the Concurrency Workbench[1]. In section 3 we look at *testing equivalence* and the *testing preorders* which generate it. We examine issues of nondeterminism and divergence in terms of their effect on the ordering of processes, and the way in which these properties are captured by the notion of *acceptance sets.* In section 4 we look briefly at the automated CCS tool, the *Concurrency Workbench.*

4.1 Operational view

CCS agents are generally understood in terms of *transition semantics* [Mil89]. The language is a *labelled transition system* and the semantics of all the combinators are expressed in terms of transitions, or in other words, the behaviours they induce. A

[1] All concurrency workbench environments for this chapter are included in Appendix B

labelled transition system is a triple, $(S, T, \{\xrightarrow{t}: t \in T\})$ where S is a set of states, T a set of *transition labels* and $\xrightarrow{t} \subseteq S \times S$ a transition relation for each $t \in T$. So if $S \xrightarrow{a} S'$ then S may admit the action (or transition) a, thereby evolving to a state S'.

This notation and semantics captures our intuition about the behaviour of processes; but if we wish to establish whether two agents or processes are computationally equivalent then we need to understand at a higher level of abstraction precisely what behaviours we intend to regard as equivalent (or, indeed, equal). For sequential programs the question is easier to answer. We should probably say that two programs are computationally equivalent if, for any given input, they both give the same output; a sequential program may be regarded as a function over the state space, mapping inputs to outputs. In the case of concurrent communicating sytems, notions such as input and output are less well-defined; a degree of nondeterminism induced by internal communication between processes is almost inevitable. This suggests that for any given input there may be a range of possible outputs depending upon the behaviour of the system; it is this notion of *behaviour* that we wish to capture in defining equivalence.

Several notions of equivalence of concurrent systems have been suggested, based on the idea that a process may be understood in terms of its externally visible behaviour. However, we are still left with questions such as how much internal nondeterminism we are prepared to tolerate and how far a process's potential for divergence should influence our view of its visible behaviour. Bisimulation, the CCS standard notion of equivalence, takes a fairly tolerant view of divergence compared with testing equivalence, though is less generous about internal nondeterminism.

4.2 Bisimulation

Bisimulation is a cornerstone, as Milner describes it, of the theory of CCS, though it was actually discovered by David Park [Par80] shortly after the publication of [Mil80]. *Strong* bismulation, which is a congruence, treats the silent action τ in the same way as the visible actions. *Weak* bisimulation acknowledges the difference between τ and other actions and characterizes observation equivalence; it is not a congruence.

4.2.1 Strong bisimulation

We begin with the definition from [Mil89]:

Definition 4.1: A binary relation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ over agents is a *strong bisimulation*, denoted by $\sim$, if $(P, Q) \in \mathcal{S}$ implies, for all $\alpha \in Act$,
(i) Whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in \mathcal{S}$
(ii) Whenever $Q \xrightarrow{\alpha} Q'$ then, for some P', $P \xrightarrow{\alpha} P'$ and $(P', Q') \in \mathcal{S}$
This is a very strict relation. It is an equivalence relation (in fact, it is a congruence) which distinguishes between agents that for most purposes we would wish to be identified. Consider P and Q defined as $P \stackrel{def}{=} \alpha.\beta.0$, $Q \stackrel{def}{=} \alpha.\tau.\beta.0$; in strong bisimulation, every τ action must be matched between P and Q, so in this example, $P \not\sim Q$. In general this distinction is undesirable; in the case of Q, after α has acted we might have to 'wait' for the agent to admit β but there is no doubt about the next visible action. So in terms of visible actions, then, the two agents are indistinguishable. For this reason—that such undesirable distinctions are made—Milner rejects strong bisimulation as a general notion of equivalence.

4.2.2 Weak bisimulation and observation equivalence

First we recall the definition of the transition relation $P \stackrel{\hat{\alpha}}{\Rightarrow} P'$ for $\alpha \in Act$. The double arrow indicates that although the only visible action taking part in the transition is α, there may be any number of silent actions either before or after α taking part in the evolution of P to P'. The hat on α has the effect of removing all occurrences of τ that might be interleaved with α. What is being said when this relation is used in the definition below is as follows: when we are trying to match an α derivative of P with a corresponding evolution of Q, we don't mind how many τ actions are interleaved with α in order to get to Q'—we are willing to ignore them all: when we are trying to match a τ derivative of P, then *zero or more* τ actions of Q will satisfy.

Definition 4.2: A binary relation $\mathcal{S} \in \mathcal{P} \times \mathcal{P}$ over agents is a (weak) *bisimulation* if $(P, Q) \in \mathcal{S}$ implies, for all $\alpha \in Act$,
(i) Whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \stackrel{\hat{\alpha}}{\Rightarrow} Q'$ and $(P', Q') \in \mathcal{S}$
(ii) Whenever $Q \xrightarrow{\alpha} Q'$ then, for some P', $P \stackrel{\hat{\alpha}}{\Rightarrow} P'$ and $(P', Q') \in \mathcal{S}$

Bisimilarity can be shown to be an equivalence relation (denoted by $\approx$ and referring to the largest such relation); that is,

$P \approx Q$ if $\exists$ bisimulation $\mathcal{S}$ such that $(P, Q) \in \mathcal{S}$

Bisimulation equivalence is not in general a congruence unless we impose an extra condition of stability[2] on agents; a stable agent is one which has no leading τ's. So considering the example given in 4.2.1, $\alpha.\beta.0 \approx \alpha.\tau.\beta.0$ and since both are stable, $\alpha.\beta.0 = \alpha.\tau.\beta.0$. The full congruence is defined as follows:

Definition 4.3: P and Q are *equal* or *(observation-)congruent*, written $P = Q$, if for all α

(i) Whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \stackrel{\hat{\alpha}}{\Rightarrow} Q'$ and $P' \approx Q'$;

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

The two clauses differ from those in the previous definition only in one respect, namely, that in replacing $\stackrel{\hat{\alpha}}{\Rightarrow}$ with $\stackrel{\alpha}{\Rightarrow}$ we are insisting that each action of P is matched by at least one action of Q, and vice-versa. Note that this only applies to initial actions; after that, only bisimulation is required. This is Milner's adopted notion of equality.

Weak bisimulation alone is not a congruence as it is not preserved by summation; leading τ's in unstable agents are pre-emptive in the context of choice (though the relation is preserved by all the other operators of the calculus). The simplest example of a pair of agents which are bisimilar but not congruent is P and $\tau.P$; $P \approx \tau.P$, but the pre-emptive power of τ means that $Q + P \not\approx Q + \tau.P$.

The transition semantics of CCS do not distinguish between external and internal action in the context of $+$. In CCS,

$$P \xrightarrow{a} P' \text{ implies } P + Q \xrightarrow{a} P'$$
$$P \xrightarrow{\tau} P' \text{ implies } P + Q \xrightarrow{\tau} P'$$

that is, the internal action τ is pre-emptive; the evolution of P to P' means that Q is no longer an option for the environment. By contrast, in ATP we have

$$p \xrightarrow{a} p' \text{ implies } p + q \xrightarrow{a} p'$$
$$p \rightarrowtail p' \text{ implies } p + q \rightarrowtail p' + q$$

The internal transition affects only the state of p and does not pre-empt the choice of

[2]This condition is sufficient but not necessary since pairs of unstable agents may be bisimulation congruent.

q. Some of the problems we shall be addressing in this thesis are a direct consequence of this transition rule for CCS.

Example 1

Consider the example of a petrol filling station with a single pump. The required behaviour of the station is that petrol is delivered and then paid for; no more petrol may be served until the previous customer has paid his bill. We may specify this behaviour simply as follows:

$$FSSpec \stackrel{def}{=} \overline{deliverPetrol}.paid.FSSpec$$

An implemented petrol station consists of three agents: a petrol pump, PP, a $Till$ and a $Cashier$. The petrol pump, which is connected to the till, delivers petrol and registers the cost on the till. Payment may be in cash or by credit card. Cash payment is simple. In the case of credit payment, the till (which of course is a PC with integral modem) registers the card number and checks with a central credit reference facility. If all is well, the customer's signature secures payment; if not, the customer's IOU will do the trick. As soon as the transaction is complete the cashier resets the pump and the next customer can start to fill up. The behaviour of the these agents is given as follows:

$$PP \stackrel{def}{=} \overline{deliverPetrol}.\overline{amount}.reset.PP$$

$$Cashier \stackrel{def}{=} \overline{getcash}.paid.\overline{reset}.Cashier + \overline{getcard}.(sign.paid.\overline{reset}.Cashier + getIOU.paid.\overline{reset}.Cashier)$$

$$Till \stackrel{def}{=} HQ|Term \setminus \{ok, notok, dialup\}$$
$$Term \stackrel{def}{=} amount.(EFT + getcash.Term)$$
$$EFT \stackrel{def}{=} getcard.\overline{dialup}.(ok.\overline{sign}.Term + notok.\overline{getIOU}.Term)$$
$$HQ \stackrel{def}{=} dialup.(\overline{ok}.HQ + \overline{notok}.HQ)$$

$$FS \stackrel{def}{=} PP|Cashier|Till \setminus \{amount, getcard, getcash, sign, getIOU, reset\}$$

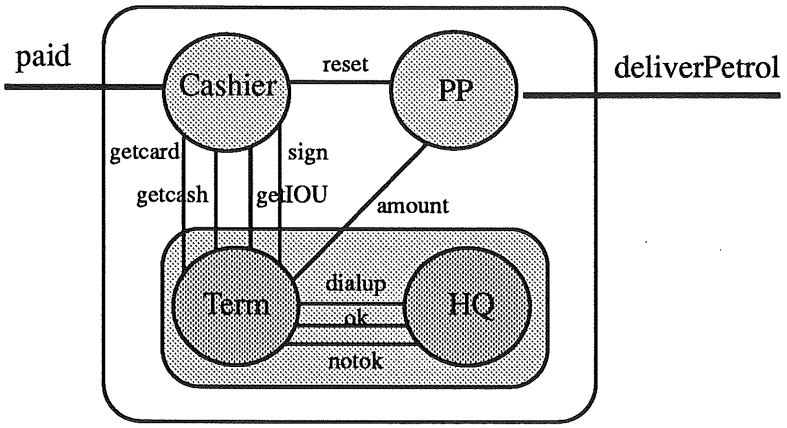

Figure 4.1: The filling station FS

We invoke the expansion law to transform the expression for FS before constructing a bisimulation between it and its specification, $FSSpec$.

$$FS = PP|Cashier|Till \setminus \{amount, getcard, getcash, sign, getIOU, reset\}$$

$$\begin{aligned} Till &= HQ|Term \setminus \{ok, notok, dialup\} \\ &= amount.((EFT + getcash.Term)|HQ) \\ &= amount.(getcash(Term|HQ) + EFT|HQ) \\ &= amount.(getcash.Till + getcard.\tau.(\tau.\overline{sign}.Till + \tau.\overline{getIOU}.Till)) \end{aligned}$$

$$\begin{aligned} FS &= PP|Cashier|Till \setminus \{amount, getcard, getcash, sign, getIOU, reset\} \\ &= \overline{deliverPetrol}.\tau.(reset.PP|Cashier|getcash.Till + \ldots) \\ &= \overline{deliverPetrol}.\tau.(\tau.paid.\tau(PP|Cashier|Till) + \tau.(\tau.\tau.paid.\tau.FS + \tau.\tau.paid.\tau.FS)) \\ &= \overline{deliverPetrol}.\tau.(\tau.paid.\tau.FS + \tau.(\tau.\tau.paid.\tau.FS + \tau.\tau.paid.\tau.FS)) \end{aligned}$$

The bisimulation between this expression and $FSSpec$ is given by $\mathcal{R}$, where

$$\begin{aligned} \mathcal{R} = \{&(FSSpec, FS), (S1, FS1), (S1, FS2), (S1, FS3), (S1, FS4), (FSSpec, FS5), \\ &(S1, FS6), (S1, FS7), (S1, FS8), (FSSpec, FS10), (FSSpec, FS11)\} \end{aligned}$$

All states after $S1$ and $FS1$ lead to the visible action *paid* (see Figure 4.2). But by invoking the equational laws in the expanded expression for FS, we can show equality of FS and $FSSpec$. The laws we invoke are:
(i) idempotence, and
(ii) $\alpha.\tau.P = \alpha.P$, the first τ law.
The equational laws of CCS are given in Appendix A.

$$\begin{aligned} FS &= \overline{deliverPetrol}.\tau.(\tau.paid.\tau.FS + \tau.(\tau.\tau.paid.\tau.FS + \tau.\tau.paid.\tau.FS)) \\ &= \overline{deliverPetrol}.\tau.(\tau.paid.FS + \tau.(\tau.paid.FS + \tau.paid.FS)) && (\tau \text{ law 1}) \\ &= \overline{deliverPetrol}.(\tau.paid.FS + \tau.(\tau.paid.FS)) && (\text{idempotence}) \\ &= \overline{deliverPetrol}.(\tau.paid.FS + \tau.paid.FS) && (\tau \text{ law 1}) \\ &= \overline{deliverPetrol}.\tau.paid.FS && (\text{idempotence}) \\ &= \overline{deliverPetrol}.paid.FS && (\tau \text{ law 1}) \end{aligned}$$

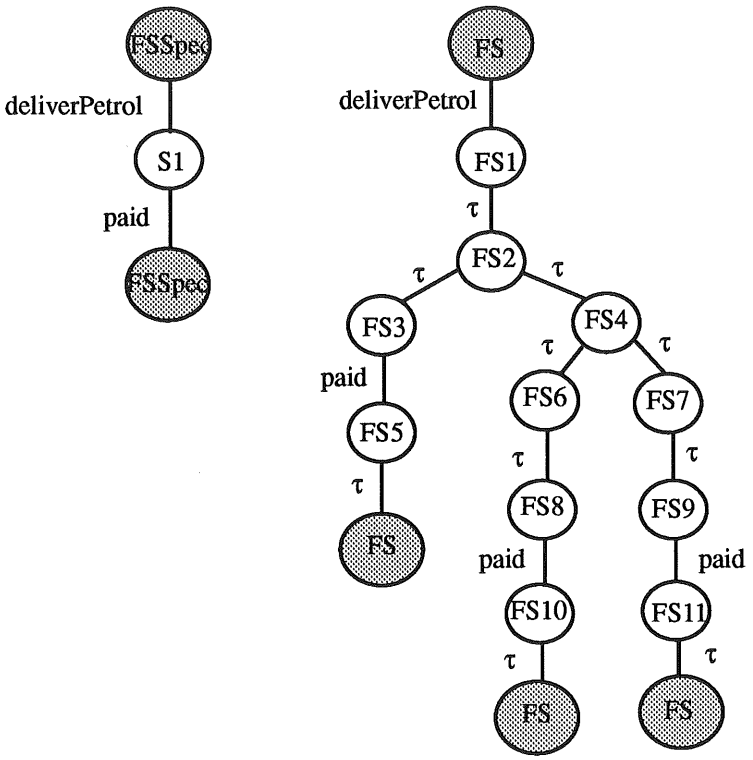

Figure 4.2: $FSSpec$ and its implementation, FS

Example 2

For our second example the bisimulation construction is less obvious. Consider the agents

$$P = a.P + \tau.b.P$$

$$Q = a.Q + \tau.(a.Q + \tau.b.Q) + \tau.b.Q$$

The bisimulation $\mathcal{R}$ is given by

$$\mathcal{R} = \{(P, Q), (P, Q_1), (P_1, Q_2), (P_1, Q_3)\}$$

(see Figure 4.3)

In the example of the filling station FS and $FSSpec$ are not only equivalent but congruent. This is not generally the case and in particular is not true of Example 2.

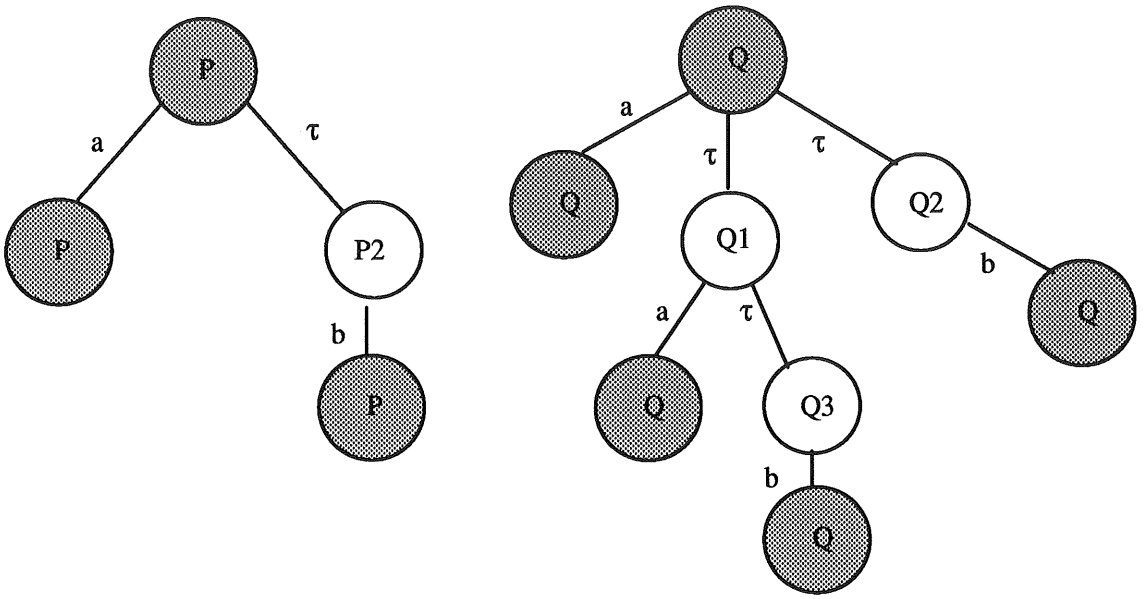

Figure 4.3: The agents P and Q

Divergence does not in itself preclude the bisimilarity of agents; for example, the agents $P \stackrel{def}{=} a.0 + \tau.P$ and $Q \stackrel{def}{=} a.0$ are bisimilar. Milner defends his equivalence on three grounds: simplicity: the ability to establish convergence by other means: and the fact that the theory allows the relative speeds of agents to vary unboundedly and so it would seem natural to allow them to vary infinitely. For example, we can define R to be the agent Q running concurrently with the divergent agent $D \stackrel{def}{=} \tau.D$, that is, $R \stackrel{def}{=} Q|D$ (see Figure 4.4); Q is convergent, and is not made less so by the fact of having D running alongside it. The visible behaviour of R is identical to that of Q so there is a strong argument in favour of equivalence. However, it is also true that R may exhibit no visible behaviour at all, an argument against permitting the equivalence of such agents.

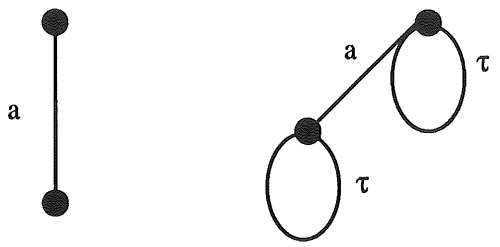

Figure 4.4: The agents Q and R

Observation equivalence

Weak bisimulation characterizes observation equivalence. Agents are identified or distinguished on the basis of their observed behaviour, which means that under some circumstances, such as the example quoted in 4.2.1, we may ignore the effect of τ actions. Intuitively, then, we might expect any two agents which are not identified under bismulation to be distinguishable by an observer or user of the system treating it as a black box. We find that this is not always the case. Consider the following pairs of agents.

Example 3

$A \stackrel{def}{=} \tau.a.0 + \tau.b.0 + \tau.(a.0 + b.0)$

$A' \stackrel{def}{=} \tau.a.0 + \tau.b.0$

In this example, informally, A' may or may not admit action a and similarly for b; it is hard to see how an observer could distinguish it from A. The observer would have to know that A was capable of moving into a state in which the choice between a and b was made by the environment. The CCS model seems to make the implicit assumption that this is the case; we might think of a and b as buttons on a machine connected to lights which came on when the appropriate choice was available. In this situation the user would always know when the choice between a and b could be made by him; but that state is one of three initial options between which the system 'chooses' nondeterministically, and that choice is hidden from the external observer. A and A' are *testing* equivalent (see 4.3).

Example 4

$P = a.0 + \tau.b.0$

$P' = \tau.b + \tau.(a.0 + b.0)$

This example is similar; we might informally describe its behaviour by saying that it always admits action b and sometimes action a. In order to infer anything more about either of these pairs of agents, the observer would need to 'open the box' and look inside. He would have to know something about the machine's hidden actions; their visible behaviour is insufficient to distinguish between them, though there is no bisimulation in either case. Again, P and P' are *testing* equivalent.

Lastly, consider the following examples; they are both specifications of an unreliable vending machine.

Example 5

$$(1) \quad VM1 = coin.\overline{drink}.VM1 + coin.BVM1$$
$$BVM1 = coin.BVM1$$

$$(2) \quad VM2 = coin.\overline{drink}.VM2 + \tau.BVM2$$
$$BVM2 = coin.BVM2$$

The first vending machine does not break until a coin is inserted; in fact, it would seem that it is precisely the insertion of the coin which causes the machine to break. The second machine breaks internally and is in the broken state *before* a coin is entered. But the only means that the observer has of finding out whether the machine is working or not is the insertion of a coin, and in each case the result is the same: either the machine is working correctly, in which case it will return a drink, or else no drink will be delivered, and the machine will be happy to accept all the coins it is offered, *ad infinitum*. As before, there is no bisimulation between $VM1$ and $VM2$, though they are testing equivalent.

In each pair of examples, there is no means by which an external observer may distinguish between the two agents; he needs 'inside knowledge' in each case. The equivalence generated by bisimulation is not truly observational; leading τ's can also *mis*lead.

4.2.3 Hennessy-Milner logic: a modal characterization of observation equivalence

An alternative method of specifying the required behaviour of processes uses a simple modal logic called *process logic*, or PL, by Milner [Mil89, HM85]. It consists of the smallest class of formulae containing the following (M in PL):

(i) T, or $true$

(ii) $\langle\alpha\rangle M$, a possibility (where $\alpha \in Act$)

(iii) $\neg M$, a negation

(iv) $\bigwedge_{i\in I} M_i$, a conjunction of formulae (I an indexing set)

So for example, if the formula $\langle\alpha\rangle\langle\beta\rangle true$ holds of a process P then, informally, it is possible for P to engage in an α experiment, thereby reaching a state P', say, which can then engage in a β experiment. This is not to say that *every* α experiment on P can be followed by a β experiment; if that property were to hold for P we could say that it is *not* possible *not* to engage in an α experiment (followed by a *beta* experiment), written $\neg\langle\alpha\rangle\neg\langle\beta\rangle\, true$. We introduce another modal operator as a shorthand for $\neg\langle\alpha\rangle\neg$, written $[\alpha]$; that is

$$[x]M \stackrel{def}{=} \neg\langle x\rangle\neg\, M$$

A satisfaction relation $\models$ is defined between formulae and processes as follows:

(i) $P \models T$

(ii) $P \models \langle\alpha\rangle M$ if, for some P', $P \stackrel{\alpha}{\rightarrow} P'$ and $P' \models M$

(iii) $P \models \neg\, M$ if $P \not\models M$ (i.e., it is not the case that $P \models M$)

(iv) $P \models \bigwedge_{i\in I} M_i$ if, for all $i \in I$, $P \models M_i$

Consider the process $P = a.b.0 + a.c.0 + b.c.0$. The following are a few of its properties (there are always alternative formulae to express a property):

	Property of P	**Modal formula: $P \models$**
1.	a successful a experiment may be followed by a successful b experiment	$\langle a \rangle \langle b \rangle \, true$
2.	*not* every successful a experiment can be followed by a successful b experiment	$\neg [a] \langle b \rangle \, true$
3.	every successful b experiment is followed by a successful c experiment	$[b][c] \, true$
4.	a c experiment is not possible	$\neg \langle c \rangle \, true$ (or $[c] \, false$)
5.	P has properties 1 and 3	$\langle a \rangle \langle b \rangle \, true \wedge [b][c] \, true$

The modal language of P, $\mathcal{L}(P)$ is the set of formulae which are true for P under $\models$. Hennessy and Milner [HM85] showed that this gives an alternative characterization of observation equivalence: two processes in CCS are observation equivalent iff they have the same modal language. A sound and complete modal characterization of observation equivalence over a small subset of CCS has been demonstrated by Stirling [Sti85]; when this subset of the language is extended to permit hidden action so that the equivalence is no longer a congruence, he gives a sound and complete proof system for observation congruence. However, there is no modal proof system for full CCS observation congruence.

Close consideration of this method of specifying the required behaviour of processes is outside the scope of this thesis; however it would seem that there are correspondences (i) between the modal operator $\langle\rangle$, the *may* preorder and the notion of safety (see below), and (ii) between the modal operator [ ], the *must* preorder and the notion of liveness. It is hoped that future work might include an examination of these relationships.

4.2.4 Simulation

This is the preorder obtained by taking 'one direction' of a bisimulation. We may define a simulation $\mathcal{S}$ as follows:

$\mathcal{S}$ is a simulation if $P\mathcal{S}Q$ implies that, for all $\alpha \in Act$,

$$\text{If } P \xrightarrow{\alpha} P' \text{ then, for some } Q',\ Q \stackrel{\hat{\alpha}}{\Rightarrow} Q' \text{ and } P'\,\mathcal{S}\,Q'$$

From this we can recover an equivalence, though the equivalence turns out *not* to be bisimulation and actually equates the two processes $P \stackrel{def}{=} a.b.0$ and $Q \stackrel{def}{=} a.b.0 + a.0$. This is not generally desirable.

4.2.5 Prebisimulation

This is a behavioural preorder based on bisimulation but taking account of divergent behaviour. First, a definition of the divergence relation $\Uparrow$ from [Wal88b]:

$$P \Uparrow \text{ iff either for some } Q, P \stackrel{\varepsilon}{\Rightarrow} Q \text{ and } Q \uparrow, \text{ or } P(\xrightarrow{\tau})^{\omega}$$

where $\uparrow$ represents infinite internal action or 'incomplete description' [Wal88b]. So that the preorder should be closed under action prefixing, it is necessary to define a *local* divergence relation $\{\Uparrow a | a \in A\}$:

$$P \Uparrow a \text{ iff either } P \Uparrow, \text{ or for some } Q, P \stackrel{a}{\Rightarrow} Q \text{ and } Q \Uparrow$$

$\Downarrow$ is the complement of this. The relation $\Uparrow a$ seems to coincide with Hennessy's '$\uparrow s$' predicate, described in 2.2.3. Prebisimulation is then defined as follows:
A relation $Pre \in P \times P$ is a *prebisimulation* if $(P,Q) \in Pre$ implies
(a) for all $u \in A \cup \{\varepsilon\}$, if $P \stackrel{u}{\Rightarrow} P'$ then for some Q', $Q \stackrel{u}{\Rightarrow} Q'$ and $(P',Q') \in Pre$
(b) for all $u \in A \cup \{\varepsilon\}$, if $P \Downarrow u$ then

(i) $Q \Downarrow u$,

(ii) if $Q \stackrel{u}{\Rightarrow} Q'$, then for some P', $P \stackrel{u}{\Rightarrow} P'$ and $(P',Q') \in Pre$.

P is prebisimilar to Q, written $P \sqsubseteq Q$ if $\exists$ prebisimulation Pre such that $(P,Q) \in Pre$
For pairs of convergent agents, prebisimulation coincides exactly with bisimulation. This relation is used in the implementation of the concurrency workbench (see 4.4).

4.3 Testing equivalence

This equivalence is due to Hennessy and de Nicola [dNH84, Hen88] and is the conjunction of two distinct equivalences: *may* equivalence, denoted by $\approx_{may}$, and *must* equivalence, denoted by $\approx_{must}$. If two processes P and Q satisfy $P \approx Q$, then $P \approx_{may} Q$ *and* $P \approx_{must} Q$. *Must* equivalence is close to Hoare's failures model—for convergent processes they coincide—and *may* equivalence coincides with the trace model [Hoa85]. Intuitively, these equivalences are constructed on the basis of tests that a process *may* (sometimes) pass or *must* (always) pass. Unlike bisimulation, testing equivalence is affected by the divergent behaviour of agents; a divergent agent can never be testing equivalent to a convergent agent.

4.3.1 The testing preorders

The testing equivalences are generated by the testing *preorders*; a preorder being reflexive and transitive means that we can recover the equivalence through the symmetric cases, that is, the kernel of the preorder.

We recall that the *may* preorder depends upon language inclusion and the *must* preorder upon both convergence and the *acceptance sets* for each node. Using the notation

$$\mathcal{L}(P) = \{s \in (Act - \{\tau\})^* | \exists P'. P \stackrel{s}{\Rightarrow} P'\}$$

for the language of P, we can define the *may* preorder as follows:

Definition 4.4:

$$P \sqsubseteq_{may} Q \text{ iff } \mathcal{L}(P) \subseteq \mathcal{L}(Q)$$

We use $S(P)$ to refer to the immediate successors of P, that is,

$$S(P) = \{a \in Act | P \stackrel{a}{\rightarrow}\}$$

The *acceptance set* of P after s is defined for CCS agents as

$$\mathcal{A}(P, s) = \{S(P') | P \stackrel{s}{\Rightarrow} P' \wedge P' \not\stackrel{\tau}{\rightarrow}\}$$

that is, $\mathcal{A}(P,s)$ is a set of the (stable) successors of P after s. This definition for CCS differs somewhat from that given in chapter 2 for Hennessy's own algebra [Hen88]. However, in each case the acceptance sets have to do with the stable, or deterministic, agents reachable from the given node and so the difference is largely cosmetic. The *must* preorder is defined as follows:

Definition 4.5:

$$P \sqsubseteq_{must} Q \text{ if} \forall s \in (Act - \{\tau\})^*, P \downarrow s \Rightarrow (Q \downarrow s \wedge \mathcal{A}(Q,s) \subset\subset \mathcal{A}(P,s))$$

where $\subset\subset$ is the relation defined in 2.2.3[3]

Example 6

Consider the process

$$P = a.b.0 + b.c.0 + a.(b.0 + c.0)$$

Its language $\mathcal{L}(P)$ is given by

$$\mathcal{L}(P) = \{\varepsilon, a, b, ab, ac, bc\}$$

Each $s \in \mathcal{L}(P)$ denotes a node of the acceptance tree for P. The acceptance sets, $\mathcal{A}(P,s)$, for each s are as follows:

$$\mathcal{A}(P,\varepsilon) = \{\{a,b\}\} \quad \mathcal{A}(P,a) = \{\{b\},\{b,c\}\} \quad \mathcal{A}(P,b) = \{\{c\}\}$$
$$\mathcal{A}(P,ab) = \{\emptyset\} \quad \mathcal{A}(p,ac) = \{\emptyset\} \quad \mathcal{A}(P,bc) = \{\emptyset\}$$

The sequence ba, for example, is not in $\mathcal{L}(P)$ and so $\mathcal{A}(P,ba) = \emptyset$, the empty set.

Example 7

Consider the two processes

$$P = a.b.0 + a.c.0$$
$$Q = a.(b.0 + c.0)$$

Here we find that $\mathcal{L}(P) = \mathcal{L}(Q) = \{\varepsilon, a, ab, ac\}$ and so immediately we have *may* equivalence (which is merely trace equivalence). The acceptance sets at each node of P are as follows:

$$\mathcal{A}(P,\varepsilon) = \{\{a\}\} \quad \mathcal{A}(P,a) = \{\{b\},\{c\}\}$$
$$\mathcal{A}(P,ab) = \{\emptyset\} \quad \mathcal{A}(P,ac) = \{\emptyset\}$$

[3] $\mathcal{A}$ and $\mathcal{B}$ satisfy $\mathcal{A} \subset\subset \mathcal{B}$ iff for every $A \in \mathcal{A}$, there exists $B \in \mathcal{B}$ such that $B \subseteq A$.

The corresponding sets for Q are

$$\mathcal{A}(Q,\varepsilon) = \{\{a\}\} \quad \mathcal{A}(Q,a) = \{\{b,c\}\}$$
$$\mathcal{A}(Q,ab) = \{\emptyset\} \quad \mathcal{A}(Q,ac) = \{\emptyset\}$$

We need only consider the acceptance sets at a since all the others are identical for each agent; we find that $\{\{b,c\}\} \subset\subset \{\{b\},\{c\}\}$ and so $P \precsim_{must} Q$. There is no bisimulation between P and Q.

Example 8

Consider the process $P = a.(b.0 + c.0) + a.b.0 + b.c.0 + a.(b.0 + \tau.c.0)$
Note that $\mathcal{A}(P,a) = \{\{b,c\},\{b\},\{c\}\}$; the last term in P gives rise to the acceptance set $\{c\}$ after a since by the definition of the acceptance set, all states represented in $\mathcal{A}(P,a)$ must be stable.

Example 9

Consider the processes $P = a.(b.P + \tau.c.P)$, $Q = a.b.Q + a.c.Q$.
Again we find that $\mathcal{L}(P) = \mathcal{L}(Q) = \{\varepsilon, a, ab, ac, aba, aca, abab, \ldots\}$, that is the processes have the same language and are therefore *may* equivalent. The acceptance sets for the first three nodes of each tree are as follows:

$$\mathcal{A}(P,\varepsilon) = \{\{a\}\} \quad \mathcal{A}(P,a) = \{\{c\}\}$$
$$\mathcal{A}(P,ab) = \{\{a\}\} \quad \mathcal{A}(P,ac) = \{\{a\}\}$$

$$\mathcal{A}(Q,\varepsilon) = \{\{a\}\} \quad \mathcal{A}(Q,a) = \{\{b\},\{c\}\}$$
$$\mathcal{A}(Q,ab) = \{\{a\}\} \quad \mathcal{A}(Q,ac) = \{\{a\}\}$$

Considering $\mathcal{A}(P,a)$ and $\mathcal{A}(Q,a)$, we find that $\mathcal{A}(P,a) \subset\subset \mathcal{A}(Q,a)$, that is, $Q \precsim_{must} P$. There is no bisimulation.

Example 10

We now consider an example of a divergent agent

$$P = a.(b.P + c.P) + \tau.P$$

and compare its behaviour to that of a similar but non-divergent agent

$$Q = a.(b.Q + c.Q)$$

Clearly P and Q are *may* equivalent. From the definition of $\sqsubseteq_{must}$, we need to compare acceptance sets at nodes where the convergence relation is satisfied; there is no such node, since one of the conditions which must hold is $P \downarrow \varepsilon$, and this is not the case for P. So $P \not\approx_{must} \Omega$; it has the potential for infinite internal computation despite its capability of stabilizing itself (temporarily) following the action a. With this in mind, we now compare P with R where

$$R = S + \tau.R \quad S = a.(b.S + c.S)$$

Again, by the Definition 4.5 of the *must* preorder, $R \downarrow \varepsilon$ does not hold and so $R \not\approx_{must} \Omega$, despite the fact that the behaviour of R is quite different from that of P. R is capable of stabilizing itself permanently once the action a is admitted; from the point of view of an experimenter, from then on its behaviour is totally predictable, but neither testing nor bisimulation distinguishes P from R (P, Q and R are bisimilar). A test to make this distinction—one that did not allow initial divergence to block all further tests—would be useful. It is not considered further here but might be the subject of future work.

Both bisimulation and testing equivalence are invoked in chapter 5 where case studies are considered. It would seem that testing equivalence gives a rather better model of externally visible behaviour, at least in the view of this writer. For reasons outlined above it is felt that the characterization (by bisimulation) of *observation* equivalence is not always true to its name. Testing equivalence is more generous and equates all the pairs of agents described in 4.2.2 which are distinguished by bisimulation. It is also a congruence[4].

[4] *must* equivalence is not a congruence over CCS agents; a and $\tau.a$ are not interchangeable under *must* testing, but $a \approx_{must} \tau.a$.

4.4 The Concurrency Workbench

To quote from the abstract of [CPS89]:

> The Concurrency Workbench is an automated tool that caters for the analysis of networks of finite-state processes expressed in Milner's Calculus of Communicating Systems. Its key feature is its scope: a variety of different verification methods, including equivalence checking, preorder checking and model checking are supported for several different process semantics.

This makes it an ideal software tool for our purposes; it is used in later chapters to establish bisimulations and the testing preorders. Without the concurrency workbench the expansion law of CCS would have to be used manually, as in the case study of chapter 3; this is a very time-consuming and error-prone business owing to the proliferation of states quite early on in the expansion, even for comparatively small systems (see [Bai91]).

The interface accepts the abstract syntax of CCS and parses it, converting it to a transition graph consisting of nodes (with one distinguished node at the root), and edges labelled by actions. Each node has associated with it an information field whose contents vary according the compuation being performed on the graph. For example, to establish *may* equivalence all that is needed is the language of the graph. This equivalence is not influenced by non-deterministic choices so a conversion is made to a deterministic graph, that is, with all internal choices removed. To establish *must* equivalence the acceptance sets at each node are required, and also information about divergence; the information fields at each node of a *must* graph, therefore, contain this information.

It is interesting to note that all the equivalences are modelled as versions of bisimulation, referred to as $\mathcal{C}$-bisimulations. If g_1 and g_2 are two transition graphs with node sets N_1 and N_2 and $N = N_1 \cup N_2$ (disjoint) then $\mathcal{C} \subseteq N \times N$ is a particular notion of equivalence, or 'compatibility of information fields' [CPS89, CH89b]. Similarly, the preorders are modelled as $\mathcal{C}$-prebisimulations [Wal88b]; in this case, $\mathcal{C}$ reflects a notion

of 'ordering on information fields'. An example of such an ordering is the $\subset\subset$ relation over acceptance sets described in 2.2.3; this relation must hold for *must* equivalence.

Chapter 5

Decomposition preordering

In this chapter we present a small case study which will be used as a vehicle for developing some of the ideas discussed in the previous chapter. The example is that of a simple level crossing, considerably less complex than the case study of chapter 3. This is for two main reasons: first, we wish to use the Concurrency Workbench[1] to test the behaviour of our specifications and, at the time of writing, parameter passing is not implemented: second, a large, complex system is not needed to illustrate the issues; the problems we will be addressing occur in quite simple examples. In addition, by considering a safety-critical system, attention is focussed on the importance of safety and liveness issues.

We begin in section 1 by looking briefly at the meaning of *safety* and *liveness* before moving on to our case study. In section 2 the system is specified at a very high level of abstraction then, in sections 3 and 4, taken through several stages of decomposition for each of two designs. Each decomposition is compared with its predecessor in order to establish either an equivalence or an ordering between them. The Concurrency Workbench is the software tool used for this purpose. We conclude that in general, the best orderings that can be established are inadequate as notions of satisfaction between specifications.

[1] All concurrency workbench environments for this chapter are included in Appendix B

5.1 Safety and liveness

Lamport [Lam83] suggests that concurrent systems may be specified in terms of their *safety* and *liveness* properties. Informally, safety is to do with the bad things a process *may not* do, and liveness with the good things it *must* do.

We consider these properties of specifications as they may be interpreted in CCS (rather than in temporal logic, as Lamport). There is no facility in CCS for defining, for example, invariant properties of a process that will guarantee its safety throughout its execution, or for translating the temporal logic assertion that a particular action must eventually occur, thereby guaranteeing liveness. Instead, we shall discuss the safety of a process in terms of the sequences of actions in which it *may* engage—that is, its traces—and liveness in terms of the sequences of actions in which it *must* engage.

5.1.1 Safety

We define safety in CCS as the prohibiting of unsafe interleavings. For example, we can specify a system in which the only visible actions are a and b and where all sequences are safe except for those containing 2 or more a's together:

$$Spec \stackrel{def}{=} a.b.Spec + b.Spec$$

We can state that *may* equivalence between any design and this specification is a proof of this particular safety property. If it has been shown that a specification contains no unsafe traces and also that a design is *may* equivalent to that specification then we can be sure that the design also is safe. Clearly any design D of which we can write $D \sqsubseteq_{may} Spec$ will also be safe *vis-à-vis* $Spec$; and since safety does not require that a process does anything at all then, theoretically, at least, this is satisfactory. To the software engineer, however—and certainly to his customer—this is quite *un*satisfactory; Ω may be safe, but it has few practical uses. We can think of *may* equivalence as a maximal safety condition.

5.1.2 Liveness

In general, liveness is the property of a system whereby every action which is required to occur does eventually do so. Given that in CCS we have no guarantee of fairness, we interpret this in its negative sense: a system is defined to be live if it is not unnecessarily prevented from making progress. Internal nondeterminism may block such progress.

The liveness condition will be stated in terms of the required behaviour (that is, successful *must* testing) of a specification under restriction; the question we shall ask in order to establish liveness is whether, when certain actions are prevented from occurring, progress *must* be made by the remaining visible actions. Internal nondeterminism may preclude this. For example, the process

$$P \stackrel{def}{=} a.P + \tau.b.P$$

is not live with respect to a; if we restrict by b the leading τ may still prevent a. It is, though, live with respect to b.

If a specification can be shown to possess a particular liveness property—that is to say, it is not unnecessarily prevented from behaving in a required manner—and *must* equivalence can be shown to hold between that (possibly restricted) specification and a (similarly restricted) design, then that design will enjoy the same liveness property. We see in the case study that the safety of a CCS implementation *vis-à-vis* its CCS specification can be established satisfactorily; in general, liveness cannot.

5.2 Specification

The level crossing admits one car or one train at a time (i.e., the car or train must leave the crossing before anything else is admitted). The signals *car_app*, *train_app* are sent by approaching vehicles; *car_cross*, *train_cross* are sent by vehicles which have been allowed to cross or, in other words, are inside a mutual exclusion zone. Note that in this model, trains always have priority; cars may be held up to allow a train (or trains) to cross, but never the other way around. This specification implicitly states the safety condition, namely, that cars and trains cannot interleave on the crossing.

$$Spec \stackrel{def}{=} car\_app.(car\_cross\ Spec + train\_app.train\_cross\ Spec') + train\_app.(train\_cross\ Spec + car\_app.train\_cross\ Spec')$$

$$Spec' \stackrel{def}{=} car\_cross\ Spec + train\_app.train\_cross\ Spec'$$

Making the substitution c for car_app, d for car_cross, u for $train\_app$ and v for $train\_cross$, we have

$$Spec \stackrel{def}{=} c.(d.Spec + u.v.Spec') + u.(v.Spec + c.v.Spec')$$

$$Spec' \stackrel{def}{=} d.Spec + u.v.Spec'$$

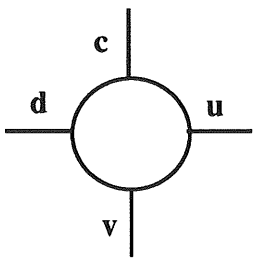

Figure 5.1: The Level Crossing

The *safety* condition may be defined in terms of legal and illegal traces; for example, *cduv*, *uvcd*, *cuvd*, *ucvd* are all legal traces. The last two may be interpreted to mean that if a car is seen to approach the crossing either immediately before or immediately after a train has been observed, the train must be allowed to proceed *before* the car. Theoretically, any number of trains may be permitted through the crossing, causing the car to wait indefinitely: $cuv(uv)^*d$ and $ucv(uv)^*d$ are all legal traces. This may be unlikely in practice but is nevertheless safe behaviour. Examples of illegal traces are *cudv* and *ucdv*; the implication of each of these is that a car has been allowed to enter the mutual exclusion zone before the train has sent the signal that it has left.

We also require that in the absence of a train, the crossing behaves like a road and similarly, in the absence of cars, like a track. That is to say, we wish that

$$Spec \setminus \{u, v\} = Road \text{ where } Road \stackrel{def}{=} c.d.Road$$

and

$$Spec \setminus \{c, d\} = Track \text{ where } Track \stackrel{def}{=} u.v.Track$$

We find that $Spec \setminus \{u, v\} = c.d\, Spec \setminus \{u, v\}$ and that $Spec \setminus \{c, d\} = u.v\, Spec \setminus \{c, d\}$, which is what is required. This is the *liveness* property.

5.3 Design 1

5.3.1 First Decomposition

We first define two agents, a road *Light* and a *Driver*; the light shows *green* if no train is approaching, i.e., it also acts implicitly as a train sensor. The *gone* signal is received when the car has crossed safely. (This would seem to imply that the train is physically stopped if a car gets stuck, though we need not be concerned with that at this level of abstraction.)

$$Light1 \stackrel{def}{=} \overline{green}.gone.Light1 + u.v.Light1$$

$$Driver \stackrel{def}{=} c.green.d.\overline{gone}.Driver$$

Then

$$D1 \stackrel{def}{=} Light1 | Driver \setminus \{gone, green\}$$

The expansion law gives

$$\begin{aligned} D1 &= c.(\tau.d.D1 + u.v.D1'_1) \\ &\quad + \\ &\quad u.(v\,D1 + c.v.D1'_1) \end{aligned}$$

$$D1'_1 = \tau.d.D1 + u.v.D1'_1$$

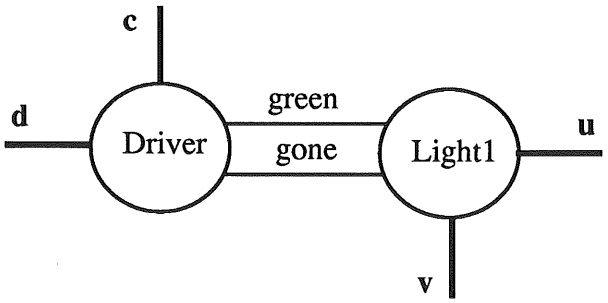

Figure 5.2: First decomposition: D1

The Concurrency Workbench gives $D1 \approx_{may} Spec$ and $D1 \sqsubseteq_{must} Spec$. The introduction of the two hidden actions *gone*, *green* has created internal nondeterminism such that bisimulation and *must* equivalence between the specification and the design are both precluded. However, $\approx_{may}$ guarantees that $D1$ cannot admit illegal traces such as *ucdv* or *cudv* (allowing cars to interleave trains) so safety is assured. This equivalence is essential; we need to be sure that the implementation allows no illegal traces. In addition, the testing preorder $\sqsubseteq$ does not imply the equivalence $\approx_{may}$ for recursive processes (though the implication does hold for finite processes); by insisting on $\approx_{may}$ we prevent the chain of preorders degenerating to Ω.

We also find that

$$D1 \setminus \{u, v\} = c.d.D1 \setminus \{u, v\}$$

and

$$D1 \setminus \{c, d\} = u.v.D1 \setminus \{c, d\}$$

that is, the liveness property is preserved.

5.3.2 Second Decomposition

The arrival of a train now explicitly changes the Light to red. Because communications are synchronized this must be 'seen' whether there is a car approaching or not; hence the need for the Observer who merely acts as a sink for this signal (in the absence of Driver).

$$Light2 \stackrel{def}{=} \overline{green}.gone.Light2 + u.\overline{red}.v.Light2$$

$$\begin{aligned} User &\stackrel{def}{=} Driver + Observer \\ Driver &\stackrel{def}{=} c.Checklights \\ Checklights &\stackrel{def}{=} green.d.\overline{gone}.User + red.Checklights \\ Observer &\stackrel{def}{=} red.User \end{aligned}$$

Then

$$D2 \stackrel{def}{=} Light2|User \setminus \{red, green, gone\}$$

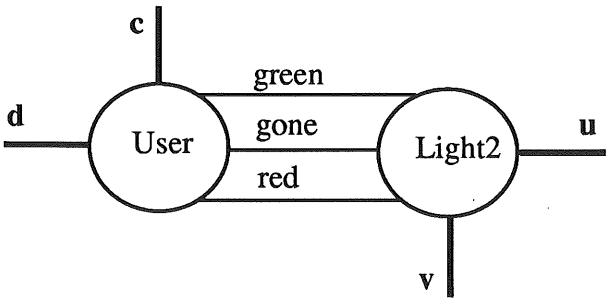

Figure 5.3: Second decomposition: D2

The output from the expansion theorem is

$$\begin{aligned} D2 &= c.(\tau.d.D2 + u.v.D2') \\ &\quad + \\ &\quad u.(\tau.v.D2 + c.v.D2') \end{aligned}$$

$$D2' = \tau.d.D2 + u.v.D2'$$

The liveness property is preserved; that is, $D2\setminus\{u, v\} = Road$ and $D2\setminus\{c, d\} = Track$. We also have $D2 \approx D1$ and $D2 = D1$ (observation congruence) from the concurrency

workbench. Since each of these equivalences is a congruence we conclude that $D1$ and $D2$ are fully interchangeable; that is, the introduction of the *red* signal, together with the necessary adjustments to *Driver* (the $User$ of $D2$) made no difference to the overall behaviour of the system. Both $D1$ and $D2$ are more non-deterministic than *Spec*; neither testing equivalence nor bisimulation holds in either case. Even so, all the desired properties of safety and liveness are preserved and so we might argue that $D1$ and $D2$ *satisfy* the specification. We continue to refine $D1$.

5.3.3 Third Decomposition

An agent $Train_a$ is introduced such that $Train_a \mid Light3$ is a decomposition of $Light1$. $Train_a$ sends the visible signals u and v to the environment, and the hidden signals *sensin* and *sensout* to the agent $Light3$.

$$Train_a \stackrel{def}{=} u.\overline{sensin}.\overline{sensout}.v.Train_a$$
$$Light3 \stackrel{def}{=} \overline{green}.gone.Light3 + sensin.sensout.Light3$$

The relationship we wish to investigate is that between $Light1$ and $TL3_a$, where

$$TL3_a \stackrel{def}{=} Train_a \mid Light3 \setminus \{sensin, sensout\}$$

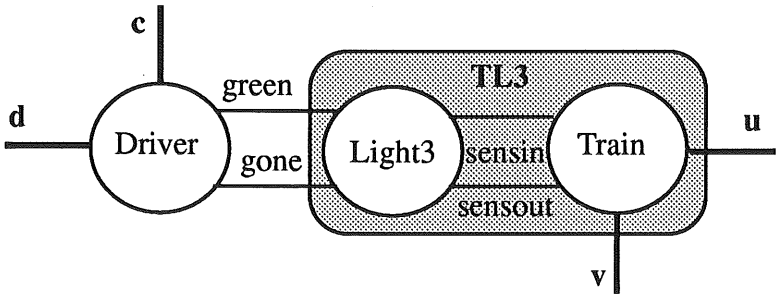

Figure 5.4: Third decomposition:D3

With $Train_a$ specified in this way, the concurrency workbench shows that $Light1$ and $TL3_a$ are not related by the *must* preorder, and that $Light1 \sqsubseteq_{may} TL3_a$. This is because of the possible interleaving of the action u with *green* and *gone*; this would

seem to be giving a car permission to cross after a train has been observed. This is sufficient to give a sequence of observable actions that imply that the system is unsafe; and when the composed system has been expanded and the actions of the sensors have been hidden, there is no guarantee of safety. It would seem that the train needs to cross the sensor (*sensin*) *before* being observed in order to be safe.

If the specification of the train is changed to

$$Train_b \stackrel{def}{=} \overline{sensin}.u.\overline{sensout}.v.Train_b$$

with

$$TL3_b = Train_b \mid Light3 \setminus \{sensin, sensout\}$$

we can see that now the train is sensed before it is observed. By the time the observer has seen the train (u), the signal *sensin* has been sent to, and received by, $Light3$, which is therefore committed to sensing the $Train_b$ leave before allowing the car to pass. The visible signal v, however, need not be sent immediately; we may think of this as the observer choosing to watch the car cross safely, and only then turning his eyes to see the train disappear into the distance. The system has operated safely, but external nondeterminism has created a different impression. As in the above case, the actions of the sensors have been hidden in the expansion and so again, no assurance of safety can be given. The preorderings between $Light1$ and $TL3_b$ are as for $Light1$ and $TL3'$, that is, $Light1 \approx_{may} TL3_b$ with no *must* ordering.

If we now change the specification of the train yet again, to

$$Train \stackrel{def}{=} \overline{sensin}.u.v.\overline{sensout}.Train$$

with $TL3 = Train \mid Light3 \setminus \{sensin, sensout\}$ so that the train is observed to leave before it is sensed by the system, we find that $TL3 \approx_{may} Light1$ and that $TL3 \not\approx_{must} Light1$. $TL3$ is not capable of interleaving the action u with *green* and *gone*, though intuitively we might feel it was operating the same way as before.

We have opened ourselves to the charge of hacking here. The difference between $Train_a$ and the other two specifications of the train is arguably an honourable one of safety; but between $Train_b$ and $Train$ we have simply altered the specification of the physical object's behaviour in order to accommodate the CCS model. This runs

counter to common sense but is, perhaps, a feature of mathematical modelling. We specify with one eye on our proof rules.

So, then, if we define

$$D3 = TL3 \mid Driver \setminus \{green, gone\}$$

we find, from the concurrency workbench, that

$$D3 \approx_{may} D1 \approx_{may} Spec$$

and

$$D3 \sqsubseteq_{must} D1 \sqsubseteq_{must} Spec$$

In the case of the third decomposition, however, we find the liveness property weakened, i.e.,

$$D3 \setminus \{c, d\} = Track$$

but

$$D3 \setminus \{u, v\} \neq Road$$

However, it is true that

$$D3 \setminus \{u, v\} \approx_{may} Road \text{ and } D3 \setminus \{u, v\} \sqsubseteq_{must} Road$$

From now on we shall use the following notation:

$$\sqsubseteq_{sat} = \approx_{may} \wedge \sqsubseteq_{must}$$

that is, $D3 \setminus \{u, v\} \sqsubseteq_{sat} Road$.

5.3.4 Fourth decomposition

We now add an agent *Control* such that $Control \mid Light4$ is a decomposition of *Light*3. *Control* receives the signals *sensin*, *sensout* from the Train. Where no train is approaching, *ok* is sent by *Control* to *Light*4 signalling that it is safe to show *green*, and the signal *gone* is now sent to *Control* and not to *Light*4.

$$Control \stackrel{def}{=} \overline{ok}.okt.gone.Control + sensin.sensout.Control$$

$$Light4 \stackrel{def}{=} ok.\overline{green}.\overline{okt}.Light4$$

Then

$$CL4 \stackrel{def}{=} Control \,|\, Light4 \setminus \{ok, okt\}$$

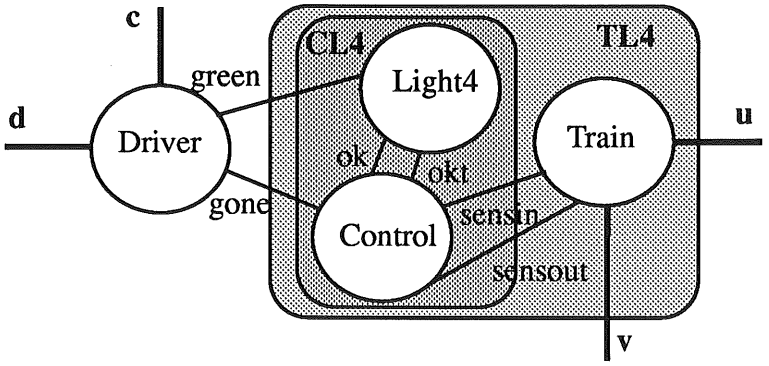

Figure 5.5: Fourth decomposition: D4

From the concurrency workbench, $CL4 \approx_{may} Light3$ and $CL4 \not\approx_{must} Light3$. If we now define $TL4 \stackrel{def}{=} CL4 \,|\, Train \setminus \{sensin, sensout\}$ then we have $TL4 \not\approx_{sat} TL3$ and

$$D4 \not\approx_{sat} D3 \text{ where } D4 \stackrel{def}{=} TL4|Driver \setminus \{green, gone\}$$

In this case the liveness property has been weakened still further. We now find that

$$D4 \setminus \{c, d\} \not\approx_{sat} Track \text{ and } D4 \setminus \{u, v\} \not\approx_{sat} Road$$

Equality has been lost through internal nondeterminism. Summarizing, the orderings are

$$D4 \approx_{may} D3 \approx_{may} D1 \approx_{may} Spec$$

and

$$D4 \not\approx_{must} D3 \not\approx_{must} D1 \not\approx_{must} Spec$$

What we have shown is that safety (*may* equivalence with $Spec$) is relatively easy to establish; liveness, however, is affected by internal nondeterminism which precludes both bisimulation and *must* equivaence. By way of comparison, we give an alternative decomposition from the same specification.

5.4 Design 2

5.4.1 First decomposition

We present a second design based on two $Controls$, one each for the train and the road traffic. The hidden communications take place between agents as follows: when a train approaches, $TControl$ sends the signal *stop* to $RControl$; when the train has been observed to leave the crossing (or when no train is approaching), $TControl$ sends *ok* (see Figure 5.6).

$$\begin{aligned} RControl1 &\stackrel{def}{=} stop.(ok.RControl1 + c.ok.RControl1') + c.RControl1' \\ RControl1' &\stackrel{def}{=} stop.ok.(RControl1' + d.RControl1) + ok.d.RControl1 \end{aligned}$$

$$TControl1 \stackrel{def}{=} \overline{stop}.u.v.\overline{ok}.TControl1 + \overline{ok}.TControl1$$

Then

$$LC1 \stackrel{def}{=} RControl1 | TControl1 \setminus \{stop, ok\}$$

From the concurrency workbench we find that $LC1 \sqsubseteq_{sat} Spec$. However, the liveness property has been weakened; we find that $LC1 \setminus \{c, d\} = Track$ (both testing and bisimulation), but $LC1 \setminus \{u, v\} \sqsubseteq_{sat} Road$.

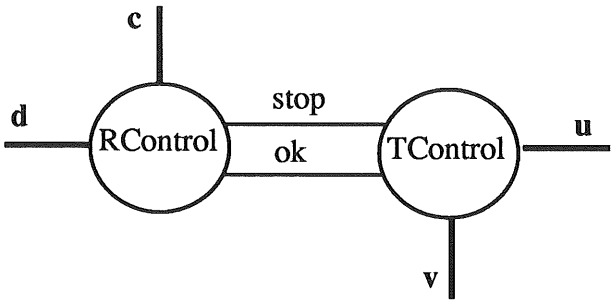

Figure 5.6: First decomposition: LC1

5.4.2 Second decomposition

We now decompose the *Controls* by introducing agents $Train$ and Car such that $TControl2 \mid Train$ is a refinement of $TControl1$ and $RControl2 \mid Car$ is a refinement of $RControl1$.

$$\begin{aligned} Car &\stackrel{def}{=} c.\overline{car_{arr}}.green.d.\overline{car_{gone}}.Car \\ RControl2 &\stackrel{def}{=} car_{arr}.(stop.ok.\overline{green}.cargone.RControl2 \\ &\qquad + ok.\overline{green}.cargone.RControl2) \\ &\quad + stop.ok.RControl2 \end{aligned}$$

Then

$$RC2 \stackrel{def}{=} Car \mid RControl2 \setminus \{car_{arr}, car_{gone}, green\}$$

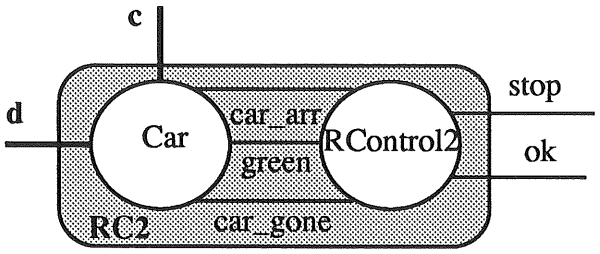

Figure 5.7: Second decomposition: RC2

$$TControl2 \stackrel{def}{=} train_{arr}.\overline{stop}.\overline{okt}.train_{gone}.\overline{ok}.TControl2 + \overline{ok}.TControl2$$
$$Train \stackrel{def}{=} \overline{train_{arr}}.okt.u.v.\overline{train_{gone}}.Train$$

and

$$TC2 \stackrel{def}{=} Train \,|\, TControl2 \setminus \{train_{arr}, train_{gone}, okt\}$$

Then

$$LC2 = TC2 \,|\, RC2 \setminus \{stop, ok\}$$

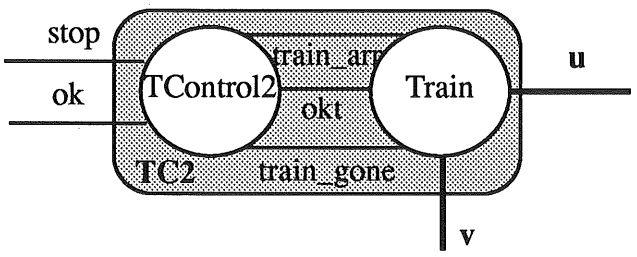

Figure 5.8: Second decomposition: TC2

The concurrency workbench shows that $TC2 \precsim_{sat} TControl1$ and $RC2 \precsim_{sat} RControl1$. $LC2 \approx LC1$ though there is no bisimulation. The liveness property of $LC2$ is as for $LC1$, i.e., $LC2\setminus\{c,d\} = Track$ and $LC2\setminus\{u,v\} \precsim_{sat} Road$.

5.5 Discussion

What has been illustrated in the above case study is the following: that when an expanded composition contains unstable agents then, in general, it is not possible to prove any equivalence with a higher specification stronger than *may* equivalence. Leading τ's often preclude the possibility of either bisimulation or *must* equivalence. The best we can hope for in general is $\underline{\subseteq}_{sat}$, the conjunction of $\underline{\simeq}_{may}$ and $\underline{\subseteq}_{must}$, together with a 'walkthrough' of the specification in order to satisfy the customer that various conditions are indeed satisfied. Where *may* equivalence with the top level specification is provable we can say with confidence that the model is safe—or at least, as safe as its specification. But the liveness condition is weakened (within the limits of the expressive power of the calculus) by internal nondeterminism. This is not satisfactory. In the next chapter we suggest an alternative; it is a conservative extension to CCS which addresses some of the problems described here and, we think, moves towards a tentative answer.

Chapter 6

Concurrent CCS

In this chapter we consider an extension to CCS, called Concurrent CCS (CCCS). It is introduced with formal semantics in [Smi91]. In the first section we give an informal introduction to the new operators and motivate this extension to CCS. In section 2 we give the transition semantics of CCCS. In section 3 we state the Concurrent Expansion Law; the proof is left till section 5. In section 4 we take another look at the case study from chapter 5 and respecify both the overall behaviour of the system and the behaviours of its components, taking advantage of the extra capabilities of the new operators. We show that previously unstable agents may be stabilized, though at a price. In section 6 we make some tentative suggestions for modelling equivalence in the extended calculus and show that the tests in the form given by Hennessy [Hen88] are inadequate for distinguishing between CCCS agents.

6.1 Introduction

CCCS is an attempt to address some of the problems identified in the case study of chapter 3 and highlighted in chapter 5. These are mainly to do with the internal nondeterminism created by unstable agents in the output from the expansion law. Some of the instability seems artificial; it can sometimes be removed by changing the order of an observable action and a restricted action within a sequence, as shown in 5.3.3. Intuitively, we feel that the behaviour of the $Train$ (and also that of the wider

system of which the train is a component) is unaffected by the order in which these actions are admitted, and yet by judicious sequencing of the actions the system can be made to 'satisfy' its specification (by which we mean $\precsim_{sat}$) where otherwise there is no ordering between them. This raises questions about how we regard the sanctity of specifications and whether we think it is permissable to rewrite (hack?) them in the light of later results.

In other circumstances, a leading τ is unavoidable since it is the result of a safety signal which must necessarily be communicated before a particular visible action can be admitted, as in 5.3.4. It may be an integral part of the safe operation of the system—precisely the required behaviour, and yet when resolved into a leading τ by the expansion law, can seem to be arbitrary and may preclude liveness of the model.

The motivation, then, for Concurrent CCS was the wish to remove such apparently artificial instability. A new action prefix operator is introduced, denoted by $\bullet$, which 'ties together' its operands so that they occur simultaneously. Since we know that we cannot *observe* simultaneity [Mur91], the notion is captured pragmatically by the following conditions: in $a \bullet b$, a and b act simultaneously and so $a \bullet b = b \bullet a$. In particular, and most significantly for our purposes in this chapter, $\tau \bullet a = a \bullet \tau = a$. This is the result that was aimed for in the design of CCCS; in the process, however, we have developed a concurrent extension to CCS which allows and, indeed, sometimes forces multiway synchronization.

Having introduced the notion of simultaneous actions (which we shall refer to as true concurrency, other notions of the term notwithstanding), we need a new parallel composition operator with a different semantics from the interleaving composition of $|$. Apart from the fact that simultaneous actions arise naturally when two or more agents act concurrently, the operator is necessary in order to prevent the deadlock which may occur due to restriction of two or more elements in a multiset of simultaneous actions. If we have a process p, say, which contains a multiset of which a and b are elements and we restrict by $\{a, b\}$, then p may need to communicate with two distinct agents in order for the restricted actions a and b both to occur. We need to develop multiway synchronization to address this. We are not concerned here with synchrony in the sense of wishing agents to proceed in lockstep, as in SCCS [Mil83]. CCCS agents

may still proceed at indeterminate relative speeds, just as CCS agents, so we do not need to make explicit the action of *waiting*. There is no assumption of a global clock; CCCS agents may proceed independently or simultaneously. We are *enabling* synchrony rather than assuming or forcing it. One consequence of this is that parallel composition in CCCS does not distribute over summation, whereas in SCCS, the *product* combinator (denoted by $\times$ and corresponding to our parallel composition) does indeed distribute over summation.

The new parallel composition operator, denoted by ||, allows its operands to run in true concurrency, so far as considerations of synchronization will permit, as well as allowing them to run independently. || will be used anywhere that we wish to admit the possibility of simultaneous execution of processes. So in the case of an agent containing $a \bullet b$ which is composed under || with other agents, a and b will always act in the same context, that is, they act simultaneously with each other and *may* act simultaneously with appropriate multisets of actions from the other agents in the composition.

6.2 Formal semantics of CCCS

6.2.1 Notation

Act_c in CCCS is the commutative group $\{Act_c, \tau, \bullet, \bar{} \}$ with identity τ. We use u, v to range over Act_c. We will use the notation $p \xrightarrow{u} p'$ to mean that p may engage in the multiset of actions u (which may be empty) and evolve to p'. For each $s \in Act_c$, we define the multiset s' as follows:
(i) if s is atomic, then $s' = \{s\}$
(ii) $\tau' = \{\}$
(iii) if $s = u \bullet v$ then $s' = u' \uplus v'$ where $\uplus$ here denotes multiset union.
So for each $s \in Act_c$, s' denotes the multiset of observable actions that occur simultaneously. Action complementation is defined as follows:

$$\{\}^c = \{\},\ \{a, b\}^c = \{\overline{a}, \overline{b}\}, \text{ where } \overline{\overline{a}} = a.$$

We need to define one more piece of notation: let u and v be any two multisets. Then

$$u \dagger v = \{s \uplus t | s \subseteq u, t \subseteq v, u - s = \{v - t\}^c\}$$

where '$-$' denotes multiset difference. (It can easily be shown that $\dagger$ is associative.) For example, consider the multisets $u = \{a, b, \overline{c}\}$, $v = \{\overline{a}, c\}$. Then

$$u \dagger v = \{\{b\}, \{a, \overline{a}, b\}, \{b, c, \overline{c}\}, \{a, \overline{a}, b, c, \overline{c}\}\}$$

The intuition here is that *either* an action and its complement are both visible, *or* they have communicated internally. For instance, the set $\{b\}$ indicates that both a and c have communicated through their respective complementary ports; the set $\{a, \overline{a}, b\}$ indicates that c has communicated internally but a and its complement are visible. So then, interpreting this as the possible actions of $P||Q$ where $P \stackrel{def}{=} a \bullet b \bullet \overline{c}$ and $Q \stackrel{def}{=} \overline{a} \bullet c$, we have

$$P \,||\, Q = b + a \bullet \overline{a} \bullet b + b \bullet c \bullet \overline{c} + a \bullet \overline{a} \bullet b \bullet c \bullet \overline{c}$$

6.2.2 Transition semantics

We use s to range over actions in Act_c, and P, Q to range over processes.

Action Prefixing

$s \bullet P \stackrel{s'}{\rightarrow} P$

Choice

$P \stackrel{u}{\rightarrow} P'$ implies $P + Q \stackrel{u}{\rightarrow} P'$

$Q \stackrel{u}{\rightarrow} Q'$ implies $P + Q \stackrel{u}{\rightarrow} Q'$

Constants

$P \stackrel{u}{\rightarrow} P'$ and $Q \stackrel{def}{=} P$ imply $Q \stackrel{u}{\rightarrow} P'$

Interleaved composition

$P \stackrel{u}{\rightarrow} P'$ implies $P|Q \stackrel{u}{\rightarrow} P'|Q$

$Q \stackrel{u}{\rightarrow} Q'$ implies $P|Q \stackrel{u}{\rightarrow} P|Q'$

$P \stackrel{u}{\rightarrow} P'$, $Q \stackrel{u^c}{\rightarrow} Q'$ implies $P|Q \stackrel{\{\}}{\rightarrow} P'|Q'$

The semantics for this operator are substantially those for pure CCS. We are effectively regarding multisets of actions as atomic units; only two-way synchronization is permitted and then only between multisets which are uniquely complementary.

Concurrent Composition

We define this by case analysis.

(i) each component of $P \,||\, Q$ may proceed independently and interact with its environment, or

(ii) agents may proceed concurrently and interact with their environment and/or with each other. Formally,

$$P \xrightarrow{u} P' \text{ implies } P \,||\, Q \xrightarrow{u} P' \,||\, Q$$

$$Q \xrightarrow{u} Q' \text{ implies } P \,||\, Q \xrightarrow{u} P \,||\, Q'$$

$$P \xrightarrow{u} P', Q \xrightarrow{v} Q' \text{ implies } P \,||\, Q \xrightarrow{r} P' \,||\, Q' \text{ for every } r \in u \dagger v.$$

Restriction

Let L be any set of actions, $\tau \notin L$. Then

$$P \xrightarrow{u} P' \text{ implies } P\backslash L \xrightarrow{u} P'\backslash L \text{ if } u \cap L = \emptyset \text{ and } u^c \cap L = \emptyset.$$

Relabelling

$$P \xrightarrow{u} P' \text{ implies } P[f] \xrightarrow{f(u)} P'[f]$$

where f is a relabelling function and $f(u)$ is the multiset $\{f(a) \mid a \in u\}$.

We will give some examples to illustrate the use of the calculus, but first we need a mechanism for expanding agents composed under $||$.

6.3 The Concurrent Expansion Law

Proposition 6.1

Let $P \equiv (P_1||P_2||\ldots||P_n)$, $n \geq 1$. Then

$$
\begin{aligned}
P \;=\; & \Sigma\{u_i.(P_1||\ldots||P_i'||\ldots||P_n)\} \text{ where } P_i \xrightarrow{u_i} P_i' \\
+\; & \Sigma\{r_2.(P_1||\ldots||P_i'||\ldots||P_j'||\ldots||P_n) \text{ for every } r_2 \in u_i \dagger u_j, \\
& \text{where } P_i \xrightarrow{u_i} P_i', P_j \xrightarrow{u_j} P_j'\} \\
+\; & \Sigma\{r_3.(P_1||\ldots||P_i'||\ldots||P_j'||\ldots||P_k'||\ldots P_n) \text{ for every } r_3 \in (u_i \dagger u_j) \dagger u_k, \\
& \text{where } P_i \xrightarrow{u_i} P_i', P_j \xrightarrow{u_j} P_j', P_k \xrightarrow{u_k} P_k'\} \\
+\; & \ldots \\
+\; & \Sigma\{r_n.(P_1'||\ldots||P_i'||\ldots||P_n') \text{ for every } r_n \in (\ldots(u_1 \dagger u_2)\ldots \dagger u_i)\ldots \dagger u_n), \\
& \text{where } P_i \xrightarrow{u_i} P_i', 1 \leq i \leq n\}
\end{aligned}
$$

Informally, this means that the P_i may run independently through their observable actions (the first line of the law) or else they may run concurrently in any combination of tuples up to and including the case where all n are running concurrently. As can be imagined, this makes for an explosion of states very early on in the expansion in cases where there are no occurrences of the $\bullet$ operator and no restriction. The operator $\bullet$, however, has the effect of reducing the number of possible states quite significantly, as we shall show.

Example 1

Consider the agents $P \stackrel{def}{=} a \bullet b.0$, $Q \stackrel{def}{=} c.\overline{b}.0$, first of all composed under $|$:

$$
\begin{aligned}
P \,|\, Q \;&=\; c.(P \,|\, \overline{b}.Q) + a \bullet b.(0 \,|\, Q) \\
&=\; c.(a \bullet b.(0|\overline{b}.Q) + \overline{b}.(P \,|\, 0)) \\
&\quad\; + a \bullet b.c.\overline{b}.(0|0) \\
&=\; c.(a \bullet b.\overline{b}.0 + \overline{b}.a \bullet b.0) \\
&\quad\; + a \bullet b.c.\overline{b}0
\end{aligned}
$$

If we restrict by b the compositon is deadlocked after the first action c, that is

$$P \,|\, Q\backslash\{b\} = c.(P \,|\, \overline{b}.Q)\backslash\{b\}$$

Considering the same agents composed with || we have

$$\begin{aligned} P \,||\, Q &= c.(P \,||\, \overline{b}.Q) + a \bullet b.(0 \,||\, Q) \\ &\quad + a \bullet b \bullet c.(0||\overline{b}.Q) \\ &= c.(a \bullet b.(0||\overline{b}.Q) + \overline{b}.(P \,||0) + a.(0||0)) \\ &\quad + a \bullet b.c.\overline{b}.(0||0) \\ &= c.(a \bullet b.\overline{b}.0 + \overline{b}.a \bullet b.0 + a.0) \\ &\quad + a \bullet b.c.\overline{b}.0 \end{aligned}$$

Restriction by b gives the result $P \,||\, Q\backslash\{b\} = c.a.0\backslash\{b\}$.

Example 2

Consider the agents $P = a \bullet b.P$ and $Q = c.\overline{b}.Q$

$$\begin{aligned} P \,|\, Q &= c.(P \,|\, \overline{b}.Q) + a \bullet b.(P \,|\, Q) \\ &= c.(a \bullet b.(P \,|\, \overline{b}.Q) + \overline{b}.(P \,|\, Q)) + a \bullet b.(P \,|\, Q) \\ &= c.(a \bullet b.a \bullet b.(P \,|\, \overline{b}.Q) + a \bullet b.\overline{b}.(P \,|\, Q) + \overline{b}.(P \,|\, Q)) + a \bullet b.(P \,|\, Q) \end{aligned}$$

Let $r = P \,|\, \overline{b}.Q$. Then

$$r = a \bullet b.r + \overline{b}.P \,|\, Q$$

So we may rewrite

$$P \,|\, Q = c.r + a \bullet b.(P \,|\, Q), \text{ where } r = a \bullet b.r + \overline{b}.P \,|\, Q$$

If we restrict by b, the composition would deadlock following the first term of the first line of the expansion, that is,

$$P \,|\, Q\backslash\{b\} = c.(P \,|\, \overline{b}.Q)\backslash\{b\}$$

a and b must act together and also synchronize with $\overline{b}$, which is not permitted by |. Now consider P and Q composed with ||:

$$\begin{aligned} P \,||\, Q &= c.(P \,||\, \overline{b}.Q + a \bullet b.(P \,||\, Q) + a \bullet b \bullet c.(P \,||\, \overline{b}.Q) \\ &= c.(a.(P \,||\, Q) + a \bullet b.(P \,||\, \overline{b}.Q) + \overline{b}.(P \,||\, Q)) + a \bullet b \bullet c.(\overline{b}.(P \,||\, Q) + a \bullet b.(P \,||\, \overline{b}.Q)) \end{aligned}$$

Without completing this expansion it can be seen that the combinations are numerous when there is no restriction. However, if we were to restrict by b the result is

$$P \,||\, Q\backslash\{b\} = c.a.(P \,||\, Q)\backslash\{b\}$$

Example 3

$P = a \bullet b.P, \quad Q = \overline{a}.c \bullet d.Q$

Again, we consider both compositions, first of all without restrictions:

$$\begin{aligned} P \mid Q &= a \bullet b.(P \mid Q) + \overline{a}.(P \mid c \bullet d.Q) \\ &= a \bullet b.(P \mid Q) + \overline{a}.c \bullet d.(P \mid Q) \end{aligned}$$

a and $\overline{a}$ cannot perform a silent communication because a occurs in the context of $a \bullet b$ in P. If we were to restrict the composition by a the system would be deadlocked.

$$\begin{aligned} P \,||\, Q &= a \bullet b.(P \,||\, Q) + b.(P \,||\, c \bullet d.Q) \\ &\quad + \overline{a}.(P \,||\, c \bullet d.Q) \\ &= a \bullet b.(P \,||\, Q) + b.(c \bullet d.(P \mid Q) + a \bullet b.(P \,||\, c \bullet d.Q) \\ &\quad + a \bullet b \bullet c \bullet d.(P||Q)) \\ &\quad + \overline{a}.(a \bullet b.(P \,||\, c \bullet d.Q) + c \bullet d.(a \bullet b.P \,||\, Q) \\ &\quad + a \bullet b \bullet c \bullet d.(P \mid Q)) \end{aligned}$$

Again we find that without restriction, the states are numerous. However, if we restrict by a the result is $P \,||\, Q \backslash \{b\} = b.c \bullet d.(P \,||\, Q) \backslash \{b\}$

Example 4

Lastly we consider a composition of four terms: $P = a.b.P$, $Q = a.\overline{b}.Q$ and two copies of $R = c \bullet \overline{a}.R$. We restrict by a and b. We find that the composition $P \mid Q \mid R \mid R \backslash \{a, b\}$ cannot proceed; it is deadlocked from the beginning. On the other hand

$$\begin{aligned} P \,||\, Q \,||\, R \,||\, R \backslash \{a, b\} &= c.(P \,||\, \overline{b}.Q \,||\, R \,||\, R) + c.(b.P \,||\, Q \,||\, R \,||\, R) \\ &\quad + c \bullet c.(b.P \,||\, \overline{b}.Q \,||\, R \,||\, R) \backslash \{a, b\} \\ &= c.c.(b.P \,||\, \overline{b}.Q \,||\, R \,||\, R) + c.c.(b.P \,||\, \overline{b}.Q \,||\, R \,||\, R) \\ &\quad + c \bullet c.(P \,||\, Q \,||\, R \,||\, R) \backslash \{a, b\} \\ &= c.c.(P \,||\, Q \,||\, R \,||\, R) + c \bullet c.(P \,||\, Q \,||\, R \,||\, R) \backslash \{a, b\} \end{aligned}$$

What we see from these examples is that composition under $||$ gives rise to a larger state space than composition with $|$. This is what we should expect, particularly where there is no restriction of actions and no use of $\bullet$. Care is needed in the design of agents which are to be allowed to proceed in true concurrency; there will clearly be undesirable 'clashes' and certain time orderings to be avoided. Agents which

will eventually become part of a truly concurrent system will need to be specified in conjunction with one another and not as more or less independent units. It can be argued that the operator | is unnecessary in CCCS; where agents are specified using the $\bullet$ prefix operator, it is unlikely that we should then wish to restrict their composition to interleaving only. However, without |, CCCS is not a conservative extension to CCS—that is, unless it is possible to embed CCS in CCCS using only the || operator for composition, though on the face of it this seems unlikely. More work is needed on the development of the calculus before we decide we can dispense with |.

6.4 Case study

We re-examine the case study from the previous chapter and make some changes in the light of the new operators described above. Now that true concurrency is permitted we can rewrite the specification to take account of the extra possibilities. For example, a train and a car might arrive at the crossing simultaneously, modelled by $c \bullet u$; we must ensure that in this case the system behaves safely. Similarly, a car may arrive simultaneously with a train leaving the crossing, that is, $c \bullet v$. Since the crossing needs to be clear before a car is allowed to cross, no other simultaneity may be permitted. So the revised specification is as follows:

$$S^* = c.(u.v.S^{*'} + d.S^*) + u.(v.S^* + c \bullet v.S^*) + u \bullet c.v.S^{*'}$$

where

$$S^{*'} = u.v.S^{*'} + d.S^*$$

Informally, this says that there are three initial possibilities: a car arrives first (then either stops for a train or is allowed to proceed): a train arrives first (and is allowed to proceed, though a car may arrive at the same time): a car and a train arrive at the same time (in which case the car must give way to the train). The agent $S^{*'}$ is invoked when a car has been held up for a train; it says that the car may have to wait indefinitely while trains continue to use the crossing, or else it may itself be allowed to cross.

6.4.1 Design 1

First decomposition

Now we consider $D1$ from chapter 3 and examine its behaviour when the hidden action *green* is tied to the visible action d; that is, when the car crosses as soon as the driver sees the green light (see Figure 6.1). Strictly speaking, these communications are not simultaneous; however, we wish to regard them as indivisible (the driver is in a great hurry and has the car in gear). So we have

$$\begin{aligned} Light1 &\stackrel{def}{=} \overline{green}.gone.Light1 + u.v.Light1 \\ Driver^* &\stackrel{def}{=} c.green \bullet d.\overline{gone}.Driver^* \end{aligned}$$

Then

$$D1^* \stackrel{def}{=} Light1 || Driver^* \backslash \{gone, green\}$$

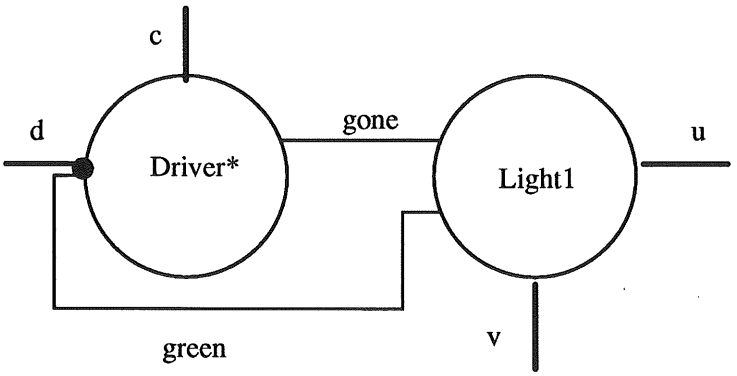

Figure 6.1: First decomposition $D1^*$

The concurrent expansion law gives

$$\begin{aligned} D1^* &= c.(Light1 || green \bullet d.\overline{gone}.Driver^*) + u.(v.Light1 || Driver^*) \\ &\quad + u \bullet c.(v.Light1 || green \ldots Driver^*) \\ &= c.(u.vD1^{*'} + d.\tau.D1^*) + u.(v.D1^* + c \bullet vD1^{*'}) + u \bullet c.v.D1^{*'}) \\ &= c.(u.vD1^{*'} + d.D1^*) + u.(v.D1^* + c \bullet vD1^*) + u \bullet c.v.D1^{*'} \end{aligned}$$

where

$$\begin{aligned} D1^{*'} &= Light1 || green \bullet d.\overline{gone} Driver^* \\ &= u.v.D1^{*'} + d.D1^* \end{aligned}$$

i.e., $D1^* = S^*$; we have equality between $D1^*$ and S^*.

Second decomposition

Now we examine $D3'$ from chapter 4 and consider the case where the observed action u is seen at precisely the same time that the system receives *sensin*; that is, we wish to tie the actions u and *sensin* (and similarly, v and *sensout*) so that they occur simultaneously.

The train is now specified as

$$Train^* \stackrel{def}{=} u \bullet \overline{sensin}.\overline{sensout} \bullet v.Train^*$$

and

$$TL3^* \stackrel{def}{=} Train^* | Light3 \backslash \{sensin, sensout\}$$

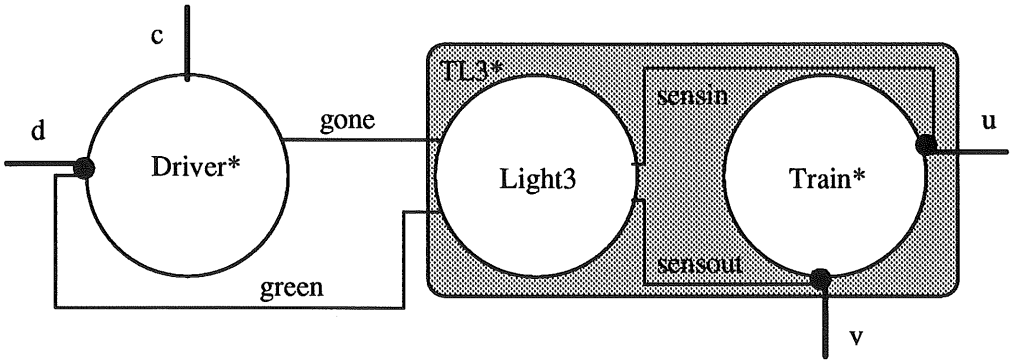

Figure 6.2: Second decomposition $D3^*$

The output from the concurrent expansion law is

$$TL3^* = u.v.TL3^{*'} + \overline{green}.gone.TL3^*$$

that is, $TL3^* = Light1$, so

$$\begin{aligned} D3^* &= Driver^* | TL3^* \backslash \{green, gone\} \\ &= Driver^* | Light1 \backslash \{green, gone\} \\ &= D1^* \quad \text{(see Figure 6.2)} \end{aligned}$$

Third decomposition

From the previous chapter we have

$$Control \stackrel{def}{=} \overline{ok}.okt.gone.Control + sensin.sensout.Control$$

$$Light4 \stackrel{def}{=} ok.\overline{green}.\overline{okt}.Light4$$

We define $Light4^*$ as

$$Light4^* \stackrel{def}{=} ok \bullet \overline{green}.\overline{okt}.Light4$$

that is, the ok signal from $Control$ causes the light to show $green$ simultaneously. Then

$$\begin{aligned} CL4^* &= Light4^* \,|\, Control\backslash\{ok, okt\} \\ &= \overline{green}.\tau.gone.CL4 + sensin.sensout.CL4\backslash\{ok, okt\} \\ &= \overline{green}.gone.CL4 + sensin.sensout.CL4\backslash\{ok, okt\} \end{aligned}$$

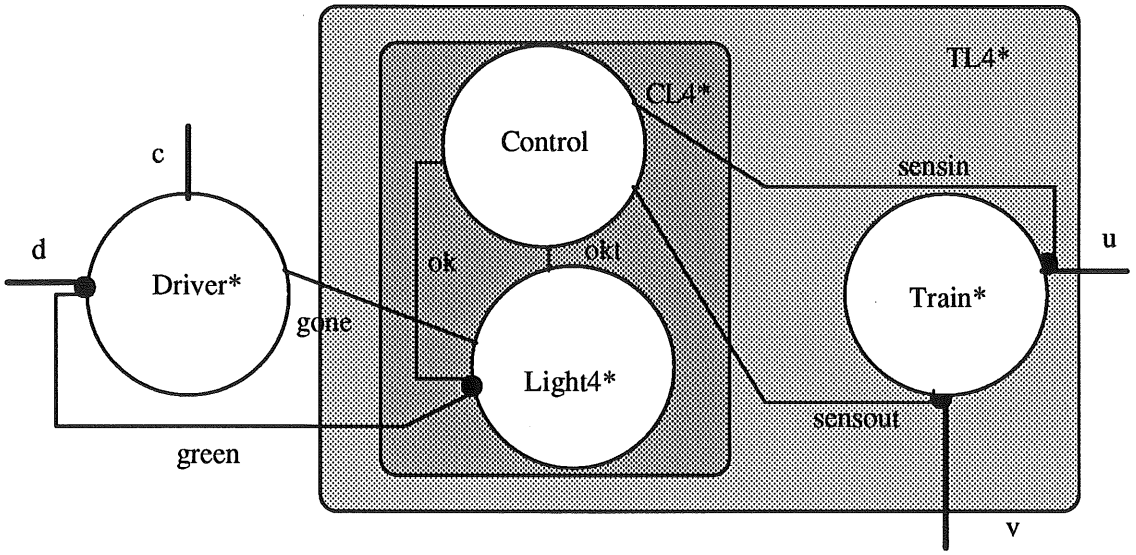

Figure 6.3: Third decomposition $D4^*$

If we now define

$$TL4^* \stackrel{def}{=} CL4^* \,||\, Train^*\backslash\{sensin, sensout\}$$

the concurrent expansion law gives

$$TL4^* = green.gone.TL4^* + uv.TL4^*$$

that is, $TL4^* = Light1$.

$D4^*$ is then

$$D4^* = Driver^* \,||\, TL4^* \backslash \{green, gone\} \quad \text{(see Figure 6.3)}$$

We have the result that $D4^* = D1^*$. For this design and the use we have made of $\bullet$, we have proved equality between each specification. This is very pleasing but probably not typical; we need to define a formal equivalence relation and also a (possibly weaker) satisfaction relation. However, previously unstable agents have been stabilized, reducing the level of internal nondeterminism. The price of this is the increased number of states, though in the above example those states represent realistic potential behaviour of a level crossing.

Next we consider the second design given in chapter 5.

6.4.2 Design 2

This is based on two control systems, one each for the road and rail traffic. Since we are allowing true concurrency we can admit more possibilities than before. The first decomposition is as follows:

$$\begin{aligned} RControl1^* &\stackrel{def}{=} stop.(ok.RControl1^* + c.ok.RControl1^{*'}) + c.RControl1^{*'} \\ &\quad + stop \bullet c.ok.RControl1^{*'} \\ RControl1^{*'} &\stackrel{def}{=} stop.ok.(RControl1^{*'} + d.RControl1^*) + ok.d.RControl1^* \end{aligned}$$

$$TControl1^* \stackrel{def}{=} \overline{stop} \bullet u.v \bullet \overline{ok}.TControl1^* + \overline{ok}.TControl1^*$$

Then

$$LC1^* \stackrel{def}{=} RControl1^* | TControl1^* \setminus \{stop, ok\}$$

We can expand this as follows:

$$\begin{aligned} LC1^* &= u.(ok.RControl1^* + c.ok.RControl1^{*'})||v \bullet \overline{ok}.TControl^*) \\ &\quad + \\ &\quad u \bullet c.(ok.RControl^{*'}||v \bullet \overline{ok}.TControl^*) \\ &\quad + \\ &\quad c.(RControl1^{*'}||TControl^*) \\ &= u.(v.LC1^* + c \bullet v.RControl1^{*'}||TControl^*) \\ &\quad + \\ &\quad u \bullet c.(v.RControl1^{*'}||TControl^*) \\ &\quad + \\ &\quad c.(RControl1^{*'}||TControl^*) \end{aligned}$$

Let $LC1^{*'} = RControl1^{*'}||TControl^*$. Then the expansion law gives

$$LC1^{*'} = u.v.LC1^{*'} + \tau.d.LC1^*$$

So the final result is

$$LC1^* = u.(v.LC1^* + c \bullet v.LC1^{*'}) + u.(v.LC1^* + c \bullet v.LC1^{*'}) + c.LC1^{*'}$$
$$\text{where } LC1^{*'} = u.v.LC1^{*'} + \tau.d.LC1^*$$

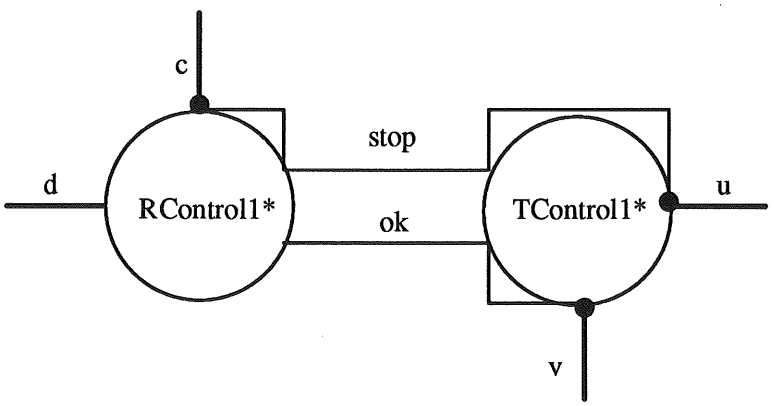

Figure 6.4: First decomposition $LC1^*$

In this case we have one unstable agent in the output, namely, the second term of $LC1^{*'}$. We could remove this if $RControl1^{*'}$ were specified as

$$RControl1^{*''} \stackrel{def}{=} stop.... + ok \bullet d.RControl1^*$$

that is to say, the car crosses at the same instant that the signal is given. This is the case in the previous design and may well be a close model of real life; it would also be very convenient here, since once again we should have equality with the specification. Purists might say that looking at the output of a proof and then altering a specification to fit was nothing more than hacking; but if the alteration is viable, that is, it models the requirements in an equally satisfactory way, then a case can be made (it's a lot cheaper than rewriting an implementation). The important point of safety here is that the car cannot proceed *before* the signal is sent that it is safe to do so, and that condition is satisfied by $RControl^{*''}$.

6.5 Proof of the Concurrent Expansion Law

We recall the law from 6.3.

Proposition 6.1

Let $P \equiv (P_1||P_2||\ldots||P_n)$, $n \geq 1$. Then

$$
\begin{aligned}
P \;=\; & \Sigma\{u_i.(P_1||\ldots||P_i'||\ldots||P_n) : P_i \xrightarrow{u_i} P_i'\} \\
+\; & \Sigma\{r_2.(P_1||\ldots||P_i'||\ldots||P_j'||\ldots||P_n) : \\
& \quad \text{for every } r_2 \in u_i \dagger u_j, \text{ where } P_i \xrightarrow{u_i} P_i', P_j \xrightarrow{u_j} P_j'\} \\
+\; & \Sigma\{r_3.(P_1||\ldots||P_i'||\ldots||P_j'||\ldots||P_k'||\ldots P_n) : \\
& \quad \text{for every } r_3 \in (u_i \dagger u_j) \dagger u_k, \text{ where } P_i \xrightarrow{u_i} P_i', P_j \xrightarrow{u_j} P_j', P_k \xrightarrow{u_k} P_k'\} \\
+\; & \ldots \\
+\; & \Sigma\{r_n.(P_1'||\ldots||P_i'||\ldots||P_n') : \\
& \quad \text{for every } r_n \in ((\ldots(u_1 \dagger u_2)\ldots \dagger u_i)\ldots \dagger u_n), \text{ where } P_i \xrightarrow{u_i} P_i', 1 \leq i \leq n\}
\end{aligned}
$$

Proof

The proof is by induction on n. For $n = 1$ we need to prove that

$$P_1 = \Sigma\{u_1.P_1' : P_1 \xrightarrow{u_1} P_1'\}$$

which follows immediately. Assume the result for n and consider $R \equiv P||P_{n+1}$. From the transition rules we have that

$$\begin{aligned} R &= \Sigma\{u_i.(P'||P_{n+1}) : P \xrightarrow{u_i} P'\} \\ &\quad +\Sigma\{u_{n+1}.(P||P'_{n+1}) : P_{n+1} \xrightarrow{u_{n+1}} P'_{n+1}\} \\ &\quad +\Sigma\{r_2.(P'||P'_{n+1}) : \text{ for every } r_2 \in u \dagger u_{n+1}, \text{ where } P \xrightarrow{u} P',\ P_{n+1} \xrightarrow{u_{n+1}} P'_{n+1}\} \\ &= R_1 + R_2 + R_3 \end{aligned}$$

From the inductive hypothesis, the first two terms of R expand to give

$$\begin{aligned} R_1 + R_2 &= \Sigma\{u_i.(P_1||\ldots||P_i'||\ldots||P_n||P_{n+1}) : P_i \xrightarrow{u_i} P_i',\ 1 \le i \le n\} \\ &+ \Sigma\{r_2.(P_1||\ldots||P_i'||\ldots||P_j'||\ldots||P_n||P_{n+1}) : \text{ for every } r_2 \in u_i \dagger u_j, \\ &\quad \text{where } P_i \xrightarrow{u_i} P_i', P_j \xrightarrow{u_j} P_j'\} \\ &+ \Sigma\{r_3.(P_1||\ldots||P_i'||\ldots||P_j'||\ldots||P_k'||\ldots P_n||P_{n+1}) : \\ &\quad \text{for every } r_3 \in (u_i \dagger u_j) \dagger w, \text{ where } P_i \xrightarrow{u_i} P_i', P_j \xrightarrow{u_j} P_j', P_k \xrightarrow{w} P_k'\} \\ &+ \ldots \\ &+ \Sigma\{r_n.(P_1'||\ldots||P_i'||\ldots||P_n'||P_{n+1}) : \\ &\quad \text{for every } r_n \in (\ldots(u_1 \dagger u_2)\ldots \dagger u_i)\ldots \dagger u_n), \text{ where } P_i \xrightarrow{u_i} P_i', 1 \le i \le n\} \\ &+ \Sigma\{u_i.(P_1||\ldots||P_i'||\ldots||P_n||P_{n+1}) : P_{n+1} \xrightarrow{u_{n+1}} P'_{n+1},\} \end{aligned}$$

The first and last lines of the above can be combined, giving the required extension to the first part of the law to accommodate P_{n+1} (that is, the part that says that each agent may act independently). So far P_{n+1} has taken no part in any communication; the third term of R gives us this. For P_{n+1} to communicate with P it may be communicating with just one of the P_i, or with two of them, or three, etc., as follows:

$$\begin{aligned} R_3 = &\ \Sigma\{r_2.(P_1||\ldots||P_i'||\ldots||P_n||P'_{n+1}) : \text{ for every } r_2 \in u_i \dagger u_{n+1}, \\ &+\Sigma\{r_3.(P_1||\ldots||P_i'||\ldots||P_j'||\ldots P_n||P'_{n+1}) : \text{ for every } r_3 \in (u_i \dagger u_j) \dagger u_{n+1}, \\ &\ldots \\ &+\Sigma\{r_{n+1}.(P_1'||\ldots||P_i'||\ldots||P_n'||P'_{n+1}) : \\ &\text{for every } r_{n+1} \in ((\ldots(u_1 \dagger u_2)\ldots \dagger u_i)\ldots \dagger u_n) \dagger u_{n+1}, \\ &\text{where } P_i \xrightarrow{u_i} P_i', 1 \le i \le n+1\} \end{aligned}$$

When we place the summand R_3 together with $R_1 + R_2$ above, all the terms in R_3 except the last term are absorbed. So we have

$$
\begin{aligned}
R \;=\;& \Sigma\{u_i.(P_1||\ldots||P_i'||\ldots||P_n||P_{n+1}) : \; P_i \xrightarrow{u_i} P_i'\} \\
+\;& \Sigma\{r_2.(P_1||\ldots||P_i'||\ldots||P_j'||\ldots||P_n||P_{n+1}) : \; \text{for every } r_2 \in u_i \dagger u_j, \\
& \quad \text{where } P_i \xrightarrow{u_i} P_i', P_j \xrightarrow{u_j} P_j'\} \\
+\;& \Sigma\{r_3.(P_1||\ldots||P_i'||\ldots||P_j'||\ldots||P_k'||\ldots P_n||P_{n+1}) : \; \text{for every } r_3 \in (u_i \dagger u_j) \dagger w, \\
& \quad \text{where } P_i \xrightarrow{u_i} P_i', P_j \xrightarrow{u_j} P_j', P_k \xrightarrow{w} P_k'\} \\
+\;& \ldots \\
+\;& \Sigma\{r_n.(P_1'||\ldots||P_i'||\ldots||P_n'||P_{n+1}) : \; \text{for every } r_n \in ((\ldots(u_1 \dagger u_2)\ldots \dagger u_i)\ldots \dagger u_n), \\
& \quad \text{where } P_i \xrightarrow{u_i} P_i', 1 \leq i \leq n\} \\
+\;& \Sigma\{r_{n+1}.(P_1'||\ldots||P_i'||\ldots||P_n'||P_{n+1}') : \\
& \quad \text{for every } r_{n+1} \in ((\ldots(u_1 \dagger u_2)\ldots \dagger u_i)\ldots \dagger u_n) \dagger u_{n+1}, \\
& \quad \text{where } P_i \xrightarrow{u_i} P_i', 1 \leq i \leq n+1\} \;\square
\end{aligned}
$$

The proof may be extended to accommodate restriction and relabelling. The Expansion Law of CCS remains substantially unchanged except that occurrences of α, representing atomic action, are replaced by u, representing multiset action. Internal communication may only occur between complementary multisets (which may be simply complementary atomic actions).

6.6 Equivalence in Concurrent CCS

6.6.1 Bisimulation

We need to define a notion of equivalence over CCCS agents; the model which suggests itself immediately is a generalized bisimulation.

Definition 6.1: A binary relation $\mathcal{S} \in \mathcal{P} \times \mathcal{P}$ over agents is a (weak) *bisimulation* if $(P, Q) \in \mathcal{S}$ implies, for all $u \in Act_c$,
(i) Whenever $P \xrightarrow{u} P'$ then, for some Q', $Q \stackrel{\hat{u}}{\Rightarrow} Q'$ and $(P', Q') \in \mathcal{S}$
(ii) Whenever $Q \xrightarrow{u} Q'$ then, for some P', $P \stackrel{\hat{u}}{\Rightarrow} P'$ and $(P', Q') \in \mathcal{S}$

In fact, the more general form of the transition relation $\stackrel{\hat{u}}{\Rightarrow}$ is only necessary in the case when u is atomic; all occurrences of τ in multisets of actions are absorbed by

definition. In this model there can be no equivalence between pairs of agents where one contains simultaneity (i.e., the operator $\bullet$) and the other does not; equivalences between standard CCS agents are preserved and so the relation is a conservative extension of weak bisimulation.

This model, while seemingly a natural extension of bisimulation, is nevertheless suspect. Bisimulation models *observation* equivalence; the u referred to in the definition are multisets of actions occurring simultaneously, not all of which, therefore, can be observed. We need to permit *sets* of observers if we wish to adopt this model of equivalence. Notwithstanding, we illustrate the notion with two examples.

Example 5

Consider the agents

$P = a \bullet b.P$

$Q = a.Q$

$R = b.R$

The composition $Q||R$ yields

$$Q||R = a.b.(Q||R) + b.a.(Q||R) + a \bullet b.(Q||R),$$

clearly not equivalent to P; but if we use the signal c to synchronize Q and R as follows

$Q' = a \bullet c.Q'$

$R' = b \bullet \overline{c}.R'$

and restrict by c we have

$$Q'||R'\backslash\{c\} = a \bullet b.(Q'||R')\backslash\{c\} = P$$

Example 6

We now consider the agent $Spec$ defined as

$Spec \stackrel{def}{=} a.Spec + a \bullet a.Spec$

and implemented by Imp, defined by

$Imp \stackrel{def}{=} P||P\backslash\{b\}$, where $P \stackrel{def}{=} a.P + b.Q$, $Q \stackrel{def}{=} a.P$

The expansion of Imp gives

$$\begin{aligned} Imp &= a.(P||P) + a.(P||P) + a \bullet a.(P||P) + \tau.(Q||Q) \\ &= a.(P||P) + a \bullet a.(P||P) + \tau.(a.(P||P) + a \bullet a.(P||P)) \\ &= \tau.(a.(P||P) + a \bullet a.(P||P)) \\ &\approx Spec \end{aligned}$$

Imp is equivalent to its specification $Spec$. It is not congruent to $Spec$, owing to the instability of Imp.

6.6.2 Testing

We recall Hennessy's tests from 4.3.1. Tests of the form

$$1w + b_1(1w + \ldots + b_n(1w + a)\ldots)$$

or

$$1w + b_1(1w + b_2(1w + \ldots + b_n(a_1w + \ldots + a_kw)\ldots)$$

are sufficient to distinguish between any two processes which are not testing equivalent (but see 2.2.2). However it is not clear that there is a test to distinguish between $P = a.0 + b.0$ and $Q = a \bullet b.0$. The test $aw + bw$ is certainly passed by P and intuitively we might feel that Q ought also to pass such a test since it can deliver either a or b. In order to distinguish between them (which we clearly wish to do) tests of a different form need to be designed.

Related to this question, we might ask how we are to extend the notion of traces to include simultaneity and what sort of trees might be the denotation of CCCS terms. We suggest the following model, exemplified by the agent $P \stackrel{def}{=} a \bullet b \bullet c.d.0$ in Figure 6.5:

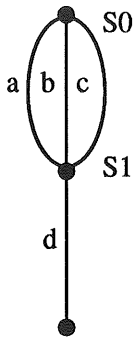

Figure 6.5: The tree for the CCCS agent $Q \stackrel{def}{=} a \bullet b \bullet c.d.0$

What we see from this is that in order for P to evolve from the state S_0 to the state S_1 all three actions a, b and c must have occurred. However the observer can take only one path through the tree, though he has three choices for this, namely, ad, bd, cd. This suggests trace (or *may*) equivalence with the agent $Q \stackrel{def}{=} ad + bd + cd$, represented by the tree shown in Figure 6.6.

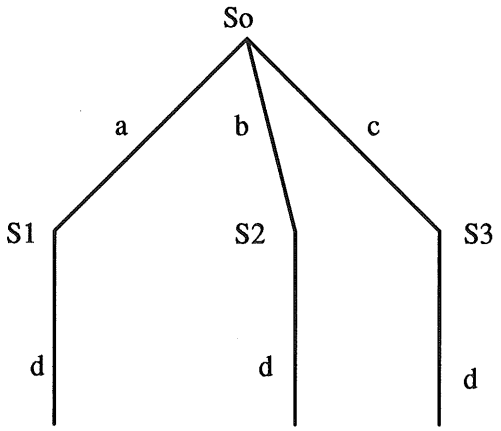

Figure 6.6: The tree for the CCCS agent $Q \stackrel{def}{=} ad + bd + cd$

Indeed we may well not wish to distinguish between these two in the trace model; an experimenter who can only observe one thing at a time (in keeping with the relativistic view) is subject to severe limitations on his capability. In that case, it is intuitively right to identify P and Q in this model. We then have that

$$\mathcal{L}(P) = \mathcal{L}(Q) = \{\varepsilon, a, b, c, ad, bd, cd\} \quad *$$

However, if we feel that, notwithstanding the real limitations of experimenters, this

identification is too generous and we wish to distinguish between processes such as P and Q then we might express the language of P as

$$\mathcal{L}'(P) = \{\varepsilon, \{a,b,c\}, \{a,b,c\}d\} \quad **$$

to suggest the choice of any element of the set $\{a,b,c\}$ but also implying a difference from $\mathcal{L}(P)$ above. In this case, unfortunately, language inclusion is no longer the model for *may* preordering. The intrinsic weakness of the trace model together with the nice properties it enjoys lead us to think that there is no need for this distinction and to accept $\mathcal{L}(P)$ as a model of the traces of P, identifying as it does the agents P and Q. Unfortunately, this means that we need to define a different class of test for the explicit purpose of establishing *must* preordering only, and only where CCCS agents are involved (the tests are redundant otherwise); we shall explore this a little further.

Extending the set of tests: restriction

In order to distinguish $P \stackrel{def}{=} a \bullet b$ from $Q \stackrel{def}{=} a + b$ we need to design tests which exploit the simultaneity of a and b in P; though it may be true that we can see either a or b (but not both) in P, we *cannot* see a without b also occurring, or b without a. This suggests tests which include *restriction*. The question whose answer will enable us to distinguish between P and Q is: can we see a if b is restricted? or b if a is restricted? We could design an experiment, e_0, such that $e_0 = aw\backslash\{b\}$; this test—and its complement $e_1 = bw\backslash\{a\}$—is passed by Q (both *may* and *must*) but not by P (neither *may* nor *must*). If we wish to retain the trace model *, then, we need to confine the use of this type of test to *must* testing only.

It is difficult to think of any test that P should pass but not Q, bearing in mind that we are interested in observable results rather than purely theoretical ones. This would give us the orderings $P \eqsim_{may} Q$ and $P \sqsubseteq_{must} Q$, with the *must* preordering here *not* relating to internal nondeterminism; in the notation of chapter 5, $P \sqsubseteq_{sat} Q$. This would be a very *un*satisfactory satisfaction relation for a vending machine that purported to give the choice of tea or coffee but, regardless of the choice, delivered both every time. The fact that the observer of the machine can see only one of the outputs will be of little comfort to the manufacturer. On the other hand, when we

visit a cinema with two screens, we choose just one to watch, though both will run concurrently regardless of our choice. In this case, the satisfaction relation might indeed be perfectly satisfactory.

We now consider the agents $r \stackrel{def}{=} a \bullet b + a$ and $s \stackrel{def}{=} a \bullet b + a + b$; to recap, we have
$P \stackrel{def}{=} a \bullet b$
$Q \stackrel{def}{=} a + b$
$R \stackrel{def}{=} a \bullet b + a$
$S \stackrel{def}{=} a \bullet b + a + b$
and the tests $e_0 = aw\backslash\{b\}$ and $e_1 = bw\backslash\{a\}$. We find that $Q\ must\ e_0$, $r\ must\ e_0$, $s\ must\ e_0$, $Q\ must\ e_1$, $s\ must\ e_1$ but $r\ must\ e_1$ giving

$$P \sqsubseteq_{must} R \sqsubseteq_{must} Q \approx_{must} S$$

and accepting the usual trace model, as * above, we have

$$P \approx_{may} R \approx_{may} Q \approx_{may} S$$

The type of test exemplified by e_0 and e_1 does not distinguish between Q and S. In the case of Q, the choice of a, say, implicitly precludes b; if we don't want b to occur, we simply choose a (or make no choice at all) and we can be sure of no b action. In S, by contrast, we should need to exclude b explicitly, otherwise s contains implicit nondeterminism. Since *must* testing is intended to detect this and order process at least partly on that basis, this suggests that we would want to distinguish between Q and S; if so, then the new tests are still insufficient.

Extending the set of tests: multiple experimenters

So far we have avoided using tests which include the operator $\bullet$; this is because the experimenter has always been assumed to be singular and therefore unable to observe simultaneous actions. In now introducing this operator into tests we make an implicit assumption that there is one experimenter to observe each simultaneous action. We should still like to keep the trace model so use these new tests only for *must* testing.

Tests of the form $a_1 \bullet a_2 w + a_1 \bullet a_2 \bullet a_3 w + \ldots + a_1 \bullet a_2 \ldots \bullet a_n w$ in addition to Hennessy's set of tests will distinguish between CCCS agents, and also between CCCS and standard

CCS agents. In particular, the test $e_2 = a \bullet bw$ will distinguish between Q and s in 6.6.2. This, alas, give us $Q \sqsubseteq_{must} S$: not what we want. Intuitively, *must* testing distinguishes between processes on the basis of nondeterminism and divergence; S contains a degree of nondeterminism which ought to be detected under *must* testing and which should result in S being below Q in the *must* preordering. We need a test which formalizes the question: can we engage in action a successfully and be sure that no other action has occurred? Tests that involve hiding do not quite answer this question. This is the subject of future work.

Chapter 7

Conclusion

What we have shown in chapters 3 and 5 is that any attempt to specify a system in CCS and then refine it by decomposition is likely to encounter difficulties in establishing satisfaction. Notwithstanding results such as those given in [Par87, Wal88a, Mil89], the presence of unstable agents after composition and expansion is the rule rather than the exception. This is likely to preclude both bisimulation and *must* equivalence with the specification, though *may* equivalence can usually be established. We can think of *may* equivalence as establishing safety *vis-á-vis* the specification (of the CCS model, of course; not necessarily of the real system it represents); *must* equivalence gives liveness.

In chapter 3 we found that by specifying the level crossing to contain a degree of internal nondeterminism we could establish bisimulation with the implementation. This begs several questions: how do we model the required kind of nondeterminism? What is the scope of the observer? How far should we permit the rewriting of specifications to fit their implementations? The answer to the last question might be to write partial, rather than total, specifications. We might express the safety requirements of the system as a set of modal formulae, say, which stated that we should never see both a train and a car in the crossing together, that trains took precedence over cars, and so on. A full modal specification is a characterization of observation equivalence; but a partial (minimal) specification might in some circumstances be adequate and less difficult to satisfy.

7.1 Nondeterminism

Internal nondeterminism—how to model it and how to interpret it—lies at the heart of the problem of establishing satisfaction. The CCS model allows a single leading τ in an agent expression to propagate nondeterminism to the whole expression. Contrast this with Hennessy's model, where internal choice is modelled as a separate operator affecting only its operands. So, for example, in the CCS agent $P \stackrel{def}{=} a+b+\ldots+m+\tau.n$, the last term has the effect of pre-empting the actions a to m, whereas in the process $p = a + b + \ldots + l + m \oplus n$, only the choice between m and n is uncertain; all the others are available to the experimenter[1]. The choice between these models is partly a matter of suitability; we need to ask ourselves exactly what we are trying to model and therefore which is more appropriate. But it is a drawback of CCS that we cannot restrict the scope of τ.

The fact that all hidden actions, whatever their nature, are resolved into τ creates problems. Whilst we may have confidence in our designs, their safety, liveness and so on, there is no way around the fact that once actions have been restricted in a CCS expansion, the only indicator of their existence is τ, and all τ's look the same; in the design, the restricted actions may be *enabling* liveness, say, whereas after expansion they may seem to be precluding it. This makes satisfaction difficult to establish.

7.2 Equivalence, preordering and satisfaction

Both bisimulation and testing are elegant theories with good intuition. However, for reasons outlined in chapter 4, the preference is for testing over bisimulation. Having said that, and despite the fact that, in general, testing is a more generous equivalence than bisimulation, its intolerance of divergence borders on the churlish; all processes with an initial capacity for divergence are *must* equated to Ω, even those with the capability of stabilizing themselves (see Example 4 in 4.3.1). The testing preorders are useful in establishing relations between processes which are not equivalent but

[1]Example $VM1'$ in 2.1.1 indicates how p can be modelled in CCS. The point being made here is *not* that the semantics of p cannot be translated into CCS, but rather that our intuition about p cannot be modelled in a 'natural' way.

whose behaviours are related; however, the direction of the preorders in CCS—the fact that the closer we get to implementation, the more nondeterministic our results—is counter-intuitive. The best we can manage in general between CCS specifications (after decomposition) is the relation $\sqsubseteq_{sat}$ as defined in chapter 5. This relation would allow, for example, the agent $Imp \stackrel{def}{=} a.b + a.c$ as a satisfactory implementation of $Spec \stackrel{def}{=} a.(b+c)$; the choice between b and c after a in $Spec$ is arguably of the essence of $Spec$, whilst Imp removes that choice and forces one of b and c on the environment. This is not satisfactory; we are forced (though with great reluctance) to conclude that for these purposes—that is, refinement of specifications by decomposition—CCS does not provide the best model.

7.3 Concurrent CCS

Concurrent CCS seeks to remove some of the internal nondeterminism described in 7.1; often such nondeterminism is the result of the time ordering of hidden and visible signals which we might feel is arbitrary and should not have such a profound effect on demonstrating satisfaction. By tying two actions together so that they always occur simultaneously we derive a commutative operation which can therefore remove instability.

The effects of permitting concurrency in this way are much more far reaching than merely the stabilizing of agents. We find that concurrent composition generates numerous states, far more than the interleaving composition of CCS. Specifying agents which are to be allowed to run in parallel needs, unsurprisingly, careful thought and circumspection.

Finding a suitable notion of equivalence over CCCS agents is difficult; traditionally we depend, whether in bisimulation or testing, on a single observer or experimenter. Nevertheless, in chapter 6 we have made some suggestions based on both a single observer and multiple observers. We believe that these are capable of being developed into a theory of equivalence for CCCS.

CCCS is similar in many ways to Milner's SCCS; Milner's product of actions ($\times$)

corresponds to our $\bullet$ operator, though the same combinator is used by Milner to compose agents, whereas we have defined a separate parallel combinator $||$ while still retaining interleaving composition. Our operator $||$ *enables* its operands to run in parallel but does not force them to do so, unlike $\times$. We also, like Milner, achieve multiway synchronization. We have retained the standard restriction mechanism; SCCS restricts *to* a subgroup of actions rather than *by* a set of actions. In addition, the motivation for CCCS was quite different. Milner set out to design a synchronous calculus and found that asynchrony could be embedded in it by designing an explicit *wait* action; CCCS, by contrast, was not seeking to model synchrony explicitly, but rather to remove internal nondeterminism. Concurrency modelling was, ironically, a side-effect of the search for stability.

7.4 Future work

In chapter 4 we suggested the close relationships between (a) safety, *may* testing and the modal operator $\langle\ \rangle$, and (b) liveness, *must* testing and the modal operator $[\ ]$. These relationships will be investigated further. We also noted in example 10 of chapter 4 that there are still some distinctions we would wish to make between standard CCS agents, which are not made by Hennessy's tests.

The work done here has cleared the way for further research into the development of a theory of equivalence over CCCS agents. This will probably mean devising an extended set of tests which will make the desirable distinctions and identifications, but without breaching the relativistic view of simultaneity.

Bibliography

[AH88] L. Aceto and M. Hennessy. Towards action-refinement in process algebras. Technical Report 3/88, University of Sussex, 1988.

[AH90] L. Aceto and M. Hennessy. Adding action-refinement to a finite process algebras. Technical Report 6/90, University of Sussex, 1990.

[AL88] Martin Abadi and Leslie Lamport. The existence of refinement mappings. In *3rd IEEE Conference on Logic in Computer Science*, pages 165–175. IEEE, 1988.

[Bai90] J. Baillie. An introduction to the algebraic specification of abstract data types. Technical Report 98, Hatfield Polytechnic, 1990.

[Bai91] Jean Baillie. A CCS case study: a safety-critical system. *Software Engineering Journal*, 6(4), 1991.

[Bun83] A. Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, 1983.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* CUP, 1990. Cambridge Tracts in Theoretical Computer Science No.18.

[CH88] Rance Cleaveland and Matthew Hennessy. Priorities in process algebras. In *3rd Annual Symposium on Logic in Computer Science*, pages 193–202. IEEE, 1988.

[CH89a] I. Castellani and M. Hennessy. Distributed bisimulations. *Journal of the ACM*, 36(4):887–911, October 1989.

[CH89b] R. Cleaveland and M. Hennessy. Testing equivalence as a bisimulation equivalence. In *Automatic Verification methods for Finite State Systems*, pages 11–23, 1989.

[CHJ86] B. Cohen, W.T. Harwood, and M.I. Jackson. *The Specification of Complex Systems.* Addison-Wesley, 1986.

[CPS89] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench. In *Automatic Verification methods for Finite State Systems*, pages 24–37, 1989.

[Den86] Tim Denvir. *Introduction to Discrete Mathematics for Software Engineering.* Macmillan, 1986.

[dNH84] R. de Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Gor87] Janusz Gorski. Formal support for development of safety related systems. In *Safety and Reliability Symposium.* Elsevier Applied Sciences, 1987.

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice–Hall International, 1985.

[Hof79] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid.* Penguin, 1979.

[Inc88] D.C. Ince. *An Introduction to Discrete Mathematics and Formal System Specification.* OUP, 1988.

[Jon80] Cliff B. Jones. *Software Development: A Rigorous Approach.* Prentice-Hall International, 1980.

[Jon86] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, 1986.

[Lam83] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming languages and systems*, 5(2):190–222, 1983.

[Lam84] Leslie Lamport. Solved problems, unsolved problems and non-problems in concurrency. *ACM SIGOPS OSR*, 4(19):34–44, 1984. Reprinted with permission from the 3rd PODC Conference Proceedings.

[Lem77] E. J. Lemmon. *An Introduction to Modal Logic.* Blackwell, 1977. Written in collaboration with Dana Scott.

[Mai87] Michael G. Main. A powerdomain primer. *Bulletin of EATCS*, (33):115–147, October 1987.

[Med69] Peter B. Medawar. *Induction and Intuition in Scientific Thought.* American Philosophical Society, 1969. Jayne Lectures for 1968.

[Mil80] R. Milner. *A Calculus of Communicating systems.* Springer Verlag, 1980. LNCS 92.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mil86] Robin Milner. A complete axiomatization for observational congruence of finite-state behaviours. Technical Report ECS-LFCS-86-8, LFCS, Department of Computer Science, University of Edinburgh, August 1986.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Mol89] Faron Moller. The nonexistence of finite axiomatizations for CCS congruences. Technical Report ECS-LFCS-89-87, Department of Computer Science, University of Edinburgh, November 1989.

[Mor90] Carroll Morgan. *Programming from Specifications.* Prentice-Hall International, 1990.

[MPW89] R. Milner, J. Parrow, and D. Walker. Mobile processes, part 1. In *Mathematical Foundations of Programming Semantics*, 1989.

[MS91] Joseph M. Morris and Roger C. Shaw, editors. *4th Refinement Workshop.* Springer-Verlag, 1991. Series Editor Professor C.J. van Rijsbergen.

[MT89] Faron Moller and Chris Tofts. A temporal calculus of communicating systems. Technical report, LFCS, Department of Computer Science, University of Edinburgh, December 1989. ECS-LFCS-89-104.

[Mur91] David Murphy. The physics of observation: A perspective for concurrency theorists. *Bulletin of the EATCS*, 44:192–200, June 1991.

[Par80] D.M.R. Park. *Concurrency and Automata on Infinite Sequences.* Springer Verlag, 1980. LNCS 104.

[Par87] Joachim Parrow. Verifying a CSMA/CD-Protocol with CCS. Technical Report ECS-LFCS-87-18, LFCS, Department of Computer Science, University of Edinburgh, January 1987.

[Sch86] D. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[Smi91] David Smith. Concurrent CCS: An introduction. Technical Report 126, Hatfield Polytechnic, 1991.

[Sti85] Colin Stirling. A proof-theoretic characterization of observation equivalence. *Theoretical Computer Science*, 39:27–45, 1985.

[Ten91] R.D. Tennent. *Semantics of Programming Languages.* Prentice-Hall International, 1991.

[Tof89] Chris Tofts. Timing concurrent processes. Technical report, LFCS, Department of Computer Science, University of Edinburgh, December 1989. ECS-LFCS-89-103.

[UK 91] UK Ministry of Defence, Directorate of Standardization. *Draft Defence Standard 00-55: Requirements for the Procurement of Safety Critical Software in Defence Equipment*, 1991.

[Wal88a] David Walker. Analysing mutual exclusion algorithms using CCS. Technical Report ECS-LFCS-88-45, LFCS, Department of Computer Science, University of Edinburgh, January 1988.

[Wal88b] D.J. Walker. Bisimulation equivalence and divergence in CCS. In *Proceedings of the third annual symposium on logic in computer science*, pages 186–192, 1988.