Fairness for Non-Interleaving Concurrency

by

Marta Z. Kwiatkowska

A thesis submitted for the degree of Doctor of Philosophy,
Faculty of Science, University of Leicester,
1989.

Moim Rodzicom

Z chaosu ład się tworzy. Ład, konieczność,
Jedyność chwili gdy bezmiar tworzywa
Sam się układa w swoją ostateczność
I woła jak się nazywa.

Julian Tuwim

Rzecz Czarnoleska

Abstract

Fairness in a non-interleaving semantic model for concurrency has been investigated. In contrast to the interleaving approach, which reduces non-sequential behaviours to a non-deterministic choice between possible interleavings of activities of concurrent processes, concurrency and causality were assumed as primitive notions. Mazurkiewicz's trace languages were chosen as behavioural representations of systems and Shields' asynchronous transition systems as their acceptors. The notion central to these two formalisms is one of causal independency, which determines trace equivalence (congruence) in the monoid of strings. Equivalence classes of strings are called traces. The quotient monoid of traces forms a poset with trace prefix ordering.

First, trace languages have been enhanced to allow for infinite traces; this was achieved by introducing trace preorder relation on possibly infinite strings. It has been shown that the extension gives rise to the domain of traces and an infinitary monoid, which specializes to the domain and the infinitary monoid of strings of Nivat's. Asynchronous transition systems have been equipped with a notion of a process structure; a variety of process structures ordered by refinement relation are possible for a given system. Each process structure determines projective preorder and equivalence relations in the monoid of strings, which are shown to coincide with the trace preorder and trace equivalence.

In this setting, a topological characterization of behavioural properties which includes safety, progress and fairness properties has been provided. Fairness properties form a subclass of infinitary progress properties that is closed under arbitrary union. Unconditional process fairness properties that are determined by process structures have been distinguished; they form a lattice with inclusion ordering. Finally, strength predicates were incorporated to allow for a variety of specific fairness properties such as weak and strong process fairness as well as equifairness and state fairness.

Acknowledgements

I would like to express my gratitude to Derek Andrews for all the support received while working on this thesis. I am especially grateful to Mike Shields for his help and guidance through non-interleaving semantic models for concurrency; I found numerous discussions with Mike very stimulating and thought-provoking. I would also like to thank Colin Stirling for finding time to discuss my research and suggest several improvements, in particular, the use of topological characterization. The comments of Amir Pnueli, Fred Schneider, Rick Thomas and Nick Measor are gratefully acknowledged. In addition, I am grateful to Professor Mazurkiewicz for stimulating my interest in trace languages while I attended his course on Concurrency given in Warsaw; no matter how many approaches I tried, I always found myself going back to his elegant abstraction of concurrency achieved through causal independency. My thanks also go to the administration of the University of Leicester and the Department of Computing Studies, especially Professor Ponter and Derek Andrews, for much needed financial support to enable me to visit other universities and attend conferences. Being professionally isolated as I have been, I appreciated every minute of the discussions that such visits afforded me. Finally, I would like to thank the staff at the Department of Computing Studies for putting up with the last few months of my writing up the thesis, and the Jagiellonian University in Kraków for granting me a sabbatical leave to complete this research.

My warmest thanks go to Paul Warren and my family in Poland for their support and encouragement. In addition to cooking and shopping, Paul also drew the diagrams and provided the technical assistance needed to print this document.

I started this research in October 1984 as a Research Scholar at the Department of Computing Studies, University of Leicester, while on a sabbatical leave from the Department of Computer Science, Jagiellonian University. Since February 1986 I have been a Lecturer at the University of Leicester.

Contents

1

Introduction

Modelling Concurrency

Sequences of system states are increasingly often used to model the behaviour of discrete systems. For sequential systems, such a sequence forms an admissible execution if it starts with an initial state, and each following state is obtained from its predecessor through an occurrence of some action. Labelled transition systems are commonly used as abstract representations of discrete systems. In the notation of labelled transition systems the occurrence of an action gives rise to a *state transition* denoted by $q \rightarrow^a q'$, where q, q' are *states* and a is an *action label*. A sequence of states models a complete execution of a sequential system if it is infinite, or if it is finite but no action could be applied in its last state. We call such sequences *non-extendable*; for a finite sequence, extending it with another action is undefined, while extending an infinite sequence with any action should have no effect.

When the behaviour of concurrent systems is modelled using state sequences, the situation becomes more complex. Such a system is a collection of concurrent agents (processes), possibly running on different physical processors in a distributed environment. One implication of the spatial distribution is that there could be no global clock; hence no centralised scheduler may be introduced. Every agent is *asynchronous* with other agents, and proceeds in steps by engaging in *local* actions as well as *communication* actions that require synchronisation possibly with a number of other agents. We assume all actions are *atomic*. It is desirable that an agent always proceeds as soon as it is ready to engage in a local action. The case of communication actions is slightly different, as the agent may be delayed until its synchronising partners are ready to communicate, but nevertheless the delay should always be *finite*. One is thus easily persuaded that, for a state sequence to be a complete execution of a concurrent system, it is necessary to exclude not only the sequences that are extendable in the sense explained above, but also those sequences that indefinitely delay some concurrent agent within a system. When considering concurrent systems, it is, therefore, necessary to add an additional constraint, called *fairness,* that execution sequences must satisfy. Fairness essentially imposes *finite*, but *unbounded*, delay on some component of the system, usually a concurrent process.

Although sequences of system states are commonly used to represent a single execution of a system (also called a *run*), different approaches are possible when grouping these executions into behavioural structures. The *interleaving* approach to defining semantics of

concurrent systems is based on the assumption that concurrency is not observable [Mil88]; hence it is justified to represent non-sequential behaviour by a choice between the possible interleavings of activities of concurrent agents. The behavioural structure used in the interleaving models is simply a *set* of execution sequences. One consequence of such an approach is that concurrency is not primitive, but it is expressible in terms of non-deterministic choice. Interleaving abstraction gives rise to very elegant algebraic models, e.g. CCS [Mil80] with standard transition system semantics and TCSP [BHR84] with failure semantics. The characteristic feature of algebraic models is that they distinguish two levels. The top level is a *language*, which provides a notation for defining processes (agents), typically allowing process variables and operators on processes such as non-deterministic choice and parallel composition. The lower level is *behavioural*, usually a labelled transition system. The main attraction of the interleaving models is that they are highly abstract and compositional; this is due to the presence of the language level, which means that the underlying transition system could be syntax-directed. The disadvantage of the interleaving approach is that fairness must be introduced as an assumption *external* to the behavioural level. Also, interleaving abstraction is not acceptable when modelling asynchronous behaviour, for example real-time systems and distributed systems.

An alternative semantic approach to modelling the behaviour of concurrent systems originates from net theory introduced by Petri, who distinguished *concurrency* and *causality* as two distinct behavioural phenomena. The term *non-interleaving* loosely refers to the class of all models based on assumptions similar to those of net theory. In the non-interleaving approach it is considered unimportant whether concurrency is observable or not; the important issue is that concurrency and causality are undoubtedly present in the behaviour of "truly" concurrent systems such as hardware components and distributed systems, and thus should be primitive in the model. The most widely known models in this class are variations of Petri nets [RoT86], but a number of models based on related principles have also been introduced, e.g. trace languages [Maz77], COSY [LSB79], event structures [Win86], asynchronous transition systems [Shi85a] [Bed87] [Shi88c], behavioural presentations [Shi88a]. The common feature of these models is that the language level is somewhat limited (for nets the "language" is graphical, rather than algebraic) and rarely compositional. The behavioural structure used in this approach is usually a partially ordered set of states. Such a behavioural structure has been criticised for being more complex, thus more difficult to use, than its interleaving counter-part. On the other hand, certain fairness assumptions are often expressible within the behavioural level, rather than as an external assumption.

The Issue of Fairness

The intuitive definition of fairness is the following:

> *In a fair computation, no component of the system should be delayed indefinitely.*

The notion of a component depends on the *level of granularity* and may differ from one formalism to another; it could mean, for example, a *process*, an *event*, a *transition*, or a *state*. Fairness notions also vary depending on their *strength*, that is a predicate which refers to some measure of the frequency with which the component must become possible before it could be guaranteed it eventually proceeds. A typical example is *weak fairness* [Par81] [LPS81] [Plo82], in which a component that is possible continuously from some point on must eventually proceed, and *strong fairness* [Par81] [LPS81] [Plo82], where a component that is possible infinitely often must proceed infinitely often.

The issue of fairness has caused much confusion recently. One reason for this state of affairs is that it is not always clear what granularity level and strength are appropriate in a particular case. As a result, a profusion of differing and frequently independent fairness notions have been formulated [Fra86]. These notions are often informal and model-specific due to a large number of models for concurrency, the inter-relationships of which have not been established yet. In addition, different concerns seem to guide the research: fairness may either be viewed as an issue to do with concurrent processes [CoS87] or non-deterministic choice [Fra86], thus giving rise to notions of fairness that are independent.

Fairness seems an intrinsically difficult issue. This is a consequence of the fact that *unfairness*, or lack of progress, is exhibited only by *infinite* behaviours (all finite behaviours are fair as a result of abstracting from relative speeds of processes). Obviously, those models that allow finite behaviours only do not adequately express fairness. It has been shown that powerful formalisms such as transfinite induction [LPS81] may be needed to verify systems under fairness assumptions. Some anomalies caused by fairness have also been pointed out, for example, in denotational semantics, ω-continuity of the concurrency operator is destroyed [Par81].

In the light of the above, it is not surprising that fairness has attracted some criticism [Dij88] [ChM88]. Also, there is tendency to leave fairness as the implementor's responsibility, rather than a property that should be established at the specification and verification stage. This is clearly unsatisfactory, as certain desirable properties of *reactive* systems [Pnu86], such as operating systems and airline reservation systems, can only be

established under given fairness assumptions. These properties are commonly referred to as *liveness* properties [Lam77] [AlS85] [Pnu86] and include program termination and guaranteed response to the request. It is thus important to understand the nature of fairness assumptions and their relationship with liveness properties.

Main Objectives of the Thesis

The object of the thesis is to formalize fairness properties in a universal setting provided by a non-interleaving semantic model for concurrency. Rather than define specific fairness properties for a chosen granularity level and strength, we would like to formulate an abstract notion of a fairness property. In other words, we shall attempt to characterize a mathematical space of behavioural properties and show that fairness properties constitute a subclass of this space. We shall also investigate the relationship of fairness and other properties. Finally, we would like to contrast the interleaving and non-interleaving approaches to concurrency as far as fairness properties are concerned.

We have chosen *trace languages* [Maz77] [Maz84a] [Maz84b] [Maz88] as a behavioural representation. Trace languages are an abstraction of concurrent behaviours determined by the relation of *causal independency*. The independency relation gives rise to trace equivalence (congruence) and trace preorder in the monoid of strings. Equivalence classes of strings are called *traces*. Trace preorder determines trace prefix ordering relation in the quotient monoid, which could be viewed as a partial order on system states visited during system execution.

As abstract representation of systems, we use *asynchronous transition systems* [Shi85a] [Bed87] [Shi88c], that is labelled transition systems enhanced with the notion of causal independency. Asynchronous transition systems have been shown to accept trace languages. Unlike other non-interleaving models, e.g. event structures [Win86], they offer a relatively high level of abstraction due to the notion of causal independency being syntactical. In future, this could be used to introduce a language level over asynchronous transition systems, thus giving rise to an asynchronous process algebra. The choice of asynchronous transition systems contributes to the universality of our approach as most models for discrete or concurrent systems have an underlying transition system structure.

Organization of the Thesis

Chapter 2 contains an extensive survey of existing trends in the research concerning fairness. It is independent of the rest of the thesis and could also be found in [Kwi88c] [Kwi89].

Chapter 3 introduces the model. We proceed systematically by introducing the basic definitions of traces and trace languages, followed by an extension of the theory with infinite traces. We then define asynchronous transition systems together with process structures. A variety of process structures ordered by the refinement relation are possible over a given system. We develop the projective preorder and equivalence relations determined by process structures and show that they coincide with trace preorder and equivalence, thus giving rise to a vector representation of traces. Finally, we discuss the properties of the set of all traces (the computation space) and the relationship of trace languages and asynchronous transition systems.

Chapter 4 derives weak, strong and unconditional process and event fairness suitable for asynchronous transition systems. We first show that a straightforward translation of fairness notions introduced in the interleaving models creates some anomalies. We then define the above-mentioned fairness notions and show they form hierarchies related by the strength of fairness and the process refinement relation. Lastly, we observe that we cannot express every fairness property by means of the notions defined so far.

In Chapter 5, as an attempt to define a more abstract notion of a fairness property, we present a topological characterization. We formally define behavioural properties of systems by introducing the topological spaces of safety, progress and fairness properties. We show that process fairness properties form an algebra closed under arbitrary union and intersection. Finally, we incorporate strength predicates which give rise to a variety of fairness notions including the previously inexpressible equifairness and state fairness.

Chapter 6 shows that Condition/Event nets determine asynchronous transition systems, thus providing them with trace semantics. Examples of safety, progress and fairness properties for Condition/Event nets are given.

2

A Survey of Fairness Notions

Fairness has been examined from different view-points and in varied semantic models, for example CCS, guarded commands, Petri nets, and automata. A classification of fairness notions has been proposed in many formalisms, which often required a suitable extension. Surprisingly, there still seems to be no general agreement on what fairness means and how it should be dealt with. One possible reason for this state of affairs is the multiplicity of semantic models used and the dependence of fairness on the intrinsic characteristics of these models, which makes it difficult to establish inter-relationships between fairness notions. It is not uncommon to encounter criticism of fairness, and the need for the powerful methods that reasoning with fairness constraints has called for, for example transfinite induction, is also questioned. This situation is clearly undesirable, and ways to provide better understanding of fairness, and perhaps a unified approach, should be sought.

This chapter reviews major issues in the area, the purpose being to present a taxonomy of notions of fairness, and to discuss the main directions taken and the implications of choosing a particular approach. The object of this review is to identify common features of fairness definitions and to examine the adequacy, or, in some cases, the failure, of the standard methods when applied to deal with fairness.

2.1. Introduction

A system is said to be correct if it satisfies a given specification, where the specification is a list of properties. In order to show that the system has a given property, we typically use an abstract model together with a set of proof rules; ideally, the list of properties should be sufficient to pronounce the system correct. The case of deterministic systems is considered relatively simple. However, it has long been recognised that this is not the case with concurrent (or non-deterministic) systems. Since such systems include airline reservation systems, operating systems etc. it is crucial that the list of properties prohibits all cases of undesirable behaviour.

Let us consider an example of an airline reservation system. The system consists of a waiting list of requests and a passenger list, each list being manipulated by a manager process capable of dealing with one customer at a time. Customers making booking requests are first registered on the waiting list, from which they enter the booking process, in some order, one by one. The booking process notifies the passenger list manager if a booking can be taken, which initiates the transfer of customer details from the waiting list to the passenger list. The system exploits concurrency both at the hardware and software level, that is all three processes mentioned here are, in fact, communicating sequential processes implemented on separate physical processors.

A desirable property of such a system would be to show that, if a customer A requests a service, then this service will eventually be provided. One might think that this would be guaranteed by the truly concurrent implementation since the processes do not have to wait for their time-slice in order to proceed. However, the reality is different. Imagine customers A, B and C are trying to book a flight. Their names are first registered on the waiting list, from which the booking process chooses A to be served and finds that the booking can be taken. As customer A is being transferred to the passenger list, he tries to book another flight but is temporarily stopped in the waiting list so that the two requests are not confused. While A is waiting for an entry to the booking process, the system has served customers B and C, who have successfully been transferred to the passenger list, and try to book another flight too. They get as far as the waiting list, after which B gains access to the booking process. In the meantime, A's transfer to the passenger list has

finally finished (there was some communications delay), but his second request cannot proceed because now the booking process is busy dealing with B. Having served B, the booking process chooses C's request. In the meantime, B decides to book a yet another flight. For some reason, B's request, rather than A's, is chosen when C has exited the booking process. However, C decides to book a yet another flight too, and gains access to the booking process just as B was leaving ... If this is allowed repeatedly *ad infinitum*, customer A's request will never be serviced because the booking process is busy servicing requests of B and C!

Esoteric though it may be, this is an example of an *unfair* behaviour of the system - unfair on customer A. Note that this is not deadlock - the system has not stopped, nor has any of its processes, when considered independently, behaved incorrectly. (In fact, we could have proved the system works properly using an invariance argument.) The only concern is that customer A has not made satisfactory progress although he was ready to do so all the time. We cannot show that A's request is eventually serviced unless we make explicit assumptions about the way booking requests are selected (so called *fairness constraints*). When considering an abstract model and the proof system necessary to show correctness with respect to such properties under fairness constraints, we find transfinite induction may be needed. It might be argued that it would be easier simply to suggest a plausible implementation - a FIFO queue or a priority system should be adequate here. But should fairness be the implementor's responsibility? Note that a cheaper implementation (that is busy waiting, or a round-robin algorithm in the multi-programming case) would admit the behaviour described in the example, so there is no guarantee that an implementor can recommend a correct solution unless, at least, a *possibility* of an unfair situation is exposed. Of course, what we should really be concerned with are ways which help *eliminate* the above, and similar, undesirable behaviours.

This is the object of the study of fairness. Its aims are to find adequate formalisms for imposing fairness constraints on computations in abstract models for concurrent (or, for that matter, non-deterministic) systems. The work in this area concentrates on building proof systems for proving properties under fairness constraints (like the "guaranteed service" mentioned here) and the development of proof techniques. Another major issue is examining the adequacy of existing formalisms when expressing fairness and investigating possible extensions. In addition, there is some research concerned with the classification (for example as a hierarchy) of fairness properties.

So far, there have been no attempts to provide a formal definition stating what a fairness property is, and, what is more, what it is not, although a variety of (often informally stated) model-specific notions have been introduced. Out of necessity, we shall discuss fairness notions through examples, rather than attempt a formal comparative study. We believe that an informal survey like this should precede any discussion on formal grounds, and hope that this would provide the necessary background and encouragement.

2.2. Defining Fairness

Before we present a taxonomy of specific fairness properties, we would like to summarize the main concerns that guided the research in the past when defining fairness.

We shall informally discuss fairness properties in concurrent and non-deterministic systems. By a system we shall mean a discrete system, which progresses from one state to another through actions (or transitions). For readability, the examples used here are in a variant of guarded commands [Hoa78]. A sequence of states with associated actions is called, for the purpose of this chapter, a *computation* (often referred to in literature as a *run*). Although runs are represented similarly in almost all semantic models, there is disagreement on how the runs should be grouped into a structure (e.g. a set, a computation tree, a partially ordered set) to represent the *behaviour* of a system, and hence to be considered a viable semantics. The presentation here will abstract from those differences, as the detailed discussion is out of scope for this chapter. Note that when considering fairness, it is necessary to include *infinite* computations as well as the finite ones.

Intuitive Definition

The motivation behind any notion of fairness seems to be to disallow infinite computations in which a system component is, for some reason, prevented from proceeding. All finite computations are fair; when infinite computations are considered, it may be necessary to distinguish computations that are fair from the unfair ones. Intuitively, fairness is a property of computations that can be expressed as follows:

> *No component of the system which becomes possible sufficiently often should be delayed indefinitely.*

This is a very general statement, which, we believe, brings together many fairness properties known from the literature. In order to obtain a specific fairness property we would need to say explicitly what we mean by a *system component*, a system component *becoming possible*, and *sufficiently often*. The first determines what it is that must be allowed to proceed, or, in other words, the *granularity level* of fairness, and the latter defines the conditions under which the component will proceed, that is the *strength* of fairness. The notion of a system component becoming possible usually depends on the kind of component, but it may be given different meaning depending on the actual model used. Examples of kinds of components, and what it means for a component to become possible, are as follows:

Component	*becomes possible if*
(concurrent) process	some action of the process is enabled
event	event can occur
synchronisation event	processes can synchronise
channel communication	processes can communicate on channel
guard	some guard in a non-deterministic program evaluates to true
transition	transition becomes enabled
state	state is immediately reachable

Since most models have an underlying transition-system structure, we shall often identify the notion of a component becoming possible with it being *enabled* (e.g. a transition) or *immediately reachable* (e.g. a state). The notion of a component becoming possible may vary depending on the semantics chosen, that is on the choice of the actual behavioural structure. Also, some semantic approaches may automatically exclude some runs as inadmissible, for example on the grounds that they do not correspond to acceptable sequentializations of a concurrent behaviour.

The following are the most common variations on the strength of fairness:

sufficiently often	*corresponding strength*
no restrictions	unconditional fairness
infinitely often	strong fairness
almost always	weak fairness

Almost always [CoS87] means always after some point in time and can also be described as *continuously from some point on* [Pnu86] (the actual interpretation may depend on the model) or *permanently* [Pnu86].

The above definition can also be rewritten in the following, perhaps more familiar, form:

> *If a system component becomes possible sufficiently often then it proceeds infinitely often.*

Note that the definition of fairness as introduced here considers each component in isolation, that is irrespective of the remaining components of the system, and does not put any specific bounds on the number of steps executed before each component makes progress. It is possible to strengthen fairness by defining it relative to a *group* of components that are jointly enabled, in the sense that each member of the group proceeds equally often [GFK86]. Another interesting class of fairness properties are *probabilistic* fairness properties [PnZ86].

Concurrency Fairness versus Fairness of Choice

It is possible to view fairness either as an issue fundamentally to do with concurrency or non-determinism. This leads to a distinction between concurrency fairness and fairness of choice. Let us consider the following two programs P and Q:

```
A =   do true → print('a') od
B =   do true → print('b') od
P =   A || B

Q =   do true → print('a')
      []  true → print('b')
      od
```

Program P is a parallel composition of two sequential processes A and B, whereas program Q is non-deterministic. Both P and Q never terminate and print infinite sequences of a's and b's. If we define fairness as a property that states that *no concurrent process should be delayed indefinitely* (*concurrency fairness*), then a computation allowing an infinite sequence of a's (denoted a^{ω}) is unfair with respect to this notion because it ignores process B (and so is any execution allowing only a finite number of b's). When considering program Q, however, we find we must consider a^{ω} fair (and, indeed, any computation allowing only a finite number of b's); program Q does not contain any concurrent processes, hence no concurrent process in Q has been indefinitely delayed!

Nothing prevents us, however, from defining fairness with respect to the choice of non-deterministic guards (*fairness of choice*). We thus say that an execution is fair with respect to choice if *no non-deterministic guard is ignored forever*. We quickly notice that, for program Q, this means disallowing computations leading to either a or b being printed only a finite number of times. In program P, on the other hand, all computations are fair because there is no non-deterministic choice in P, hence no guard has been discriminated against!

Concurrency fairness and fairness of choice (also called *fairness in selection* [Pnu86] or *conflict resolution fairness* [MOP88]) are independent notions, although they are sometimes identified. This confusion is caused by the fact that concurrency is often reduced to non-deterministic interleaving, in which case concurrency fairness corresponds to fairness of choice for a particular scheduler. Note that in order to distinguish these two notions of fairness we have to consider a formalism that allows for the distinction between concurrency and non-determinism.

Fairness of choice is discussed in [QuS83] [Fra86], whereas the main concern of [Par80] [LPS81] [CoS87] is to define fairness as an issue to do with concurrency.

Fairness and Granularity Level

Fairness properties are severely affected by the choice of granularity level. For a given system, they typically result in fairness notion that do not coincide.

Process versus Event Fairness

It is usually possible to view systems at two semantic levels: the level of events and the level of processes. This leads to two different notions of fairness, *event fairness*, in the sense that *no event should be delayed indefinitely*, and *process fairness*, in the sense that *no process should be delayed indefinitely*. These definitions depend on what constitutes an event and a process. Also, it is not always clear how to decompose the system into processes; a variety of decompositions that determine different fairness notions are possible.

Process fairness and event fairness do not, in general, coincide. The following is a simple example:

A = **do** true → print('a')
[] true → print('b')
od

C = **do** true → print('c') **od**

P = A || C

Process A repeatedly chooses between printing a and b, while process C engages in forever printing c. Assuming each print command is an event, then the only *event fair* computations of P are the ones that lead to an infinite number of each of a, b and c being printed. Given A and C as the identifiable processes, we note that computations that generate only a finite number of a's (or a finite number of b's) are *process fair*. Thus, the computations containing a finite number of a's are process fair, but not event fair. (Note that a different decomposition into processes is possible here by representing process A as a non-deterministic composition of processes A_a and A_b; event fairness then coincides with process fairness).

The above example suggests that it might be plausible to re-define the notion of an event so that event fairness coincides with process fairness (after all, it has been noted in [Plo82] that event fairness depends on what constitutes an event). One might, however, take a more orthodox view that assumes that no information about processes is available at the level of events, and thus consider process fairness and event fairness as two different (but perhaps not unrelated) notions.

Event fairness has been defined in [Plo82] for a concurrent while language. Process fairness for a (shared-memory) concurrent programming language is discussed in [LPS81] [Pnu86], and for CCS in [CoS84] [CoS87].

In the context of distributed models allowing inter-process communication, for example over channels, one can further refine the granularity of fairness [KdR83] by defining *channel fairness* and *process communication fairness* (not necessarily on the same channel). It can be shown that, together with process fairness, these notions form a strict hierarchy.

Transition versus State Fairness

Fairness is also discussed in the context of transition systems, often used as an abstract model that views a discrete system as progressing through transitions from one state to another. Here one can distinguish *transition fairness*[QuS83], in the sense that no transition that is infinitely often enabled should be ignored indefinitely, and *fair reachability from states* [QuS83], in the sense that no state that is immediately reachable infinitely often should be delayed indefinitely.

Let us consider the following example (adapted from [Fra86]). The states are identified with the value of x, the transitions are the assignment statements, and the guards determine if the transitions are enabled:

```
P =   (x := 1;
      do     x = 1   → x := 2
      []     x > 0   → x := x - 1
      od)
```

There is an infinite computation of this program (alternating the first and the second guard), which takes both transitions infinitely often, thus no transition has been ignored indefinitely. The contents of the variable x alternates between 1 and 2 in this computation. However, from the state x=1, which is visited infinitely often, it is possible to reach x=0; hence this computation would be disallowed under fair reachability from states. Now, as soon as x becomes 0, the program terminates, thus Q terminates under fair reachability from states but it does not under transition fairness constraints.

Fair reachability from states and transition fairness are independent notions. We have introduced here their *strong* versions only, although other strengths may also be considered [Fra86].

Fairness & Liveness

It has long been recognised [Lam77] that fairness affects liveness properties of programs. In [Lam77] liveness has been defined as a class of properties that state that *something good will happen* during program execution. This is in contrast with safety, which is pronounced as *nothing bad will happen.* Examples of liveness are: program termination (the "good thing" is that the program terminates) and guaranteed response (the "good thing" is that the request is handled). Fairness has a major effect on liveness properties in correctness proofs of non-deterministic or concurrent programs: we cannot prove certain programs correct with respect to a given liveness property unless we make explicit assumptions about fairness. This can be shown in the following example:

A = **do** true → print(0)
[] B!1 → stop
od

B = **do** A?x → (print(x); stop) **od**

P = A || B

Process A repeatedly chooses between printing zero and sending the value 1 to process B. Process B is always ready to receive from A, but it must wait to synchronise. Process P terminates only if A and B eventually synchronise. Note that P does not terminate if no scheduling is present; however, if we assume a "reasonable" scheduling, P will terminate. Termination of P cannot be formally proved unless we incorporate in the proof the assumption that process A will eventually choose the communication (fairness of choice) or that process B eventually proceeds (process fairness).

The term "liveness" has also been given a different meaning, for example in [ApO84]: *unless a process has terminated, it will proceed infinitely often.* We believe that this formulation is a fairness notion called *process liveness*, whereas the first meaning refers to a more general class of properties.

A topological characterization of liveness has been presented in [AlS85]. A number of proof techniques for liveness have been introduced, namely the proof lattice method [OwL82] and the well-founded sets method [GPS80] [Pnu86], where the intuitive definition of liveness is slightly different.

Negative versus Positive Approach

We have so far avoided the question of how to discriminate between fair and unfair computations. Note that this cannot be achieved (in a finite number of steps) by examining every finite prefix of them. Also, most existing models do not make any provisions for fairness; it is generally accepted that this way the model is more abstract, and therefore simpler to use.

Existing approaches for dealing with fairness can be split into two classes. The first approach, called *negative*, is to consider two semantic levels [LPS81]. The lower level admits all possible computations, whether fair or unfair. Then, at the higher level, some methods are invoked for *excluding* the computations that are unfair. An example of such a method is to introduce proof rules, one for each fairness notion, which are used to prove termination under the given constraints. Thus, unfair computations are considered irrelevant. Another approach is to extend the transition system semantics with explicit fairness restrictions [Pnu86] [ClG87]. The motivation for this approach is to produce correctness proofs with respect to a given scheduling policy, which could then be enforced at the implementation stage.

It is an interesting question if one can generate only fair computations of a given system, rather than exclude the unfair ones. This is the essence of the *positive* approach, in which we are dealing with one semantic level. Such an approach has been developed in [CoS84] [CoS87] for CCS through extending the CCS calculus into labelled calculus and modifying the proof rules. Other positive approaches, based on random assignment, include [Plo82] [Par81]. The positive approach could be viewed as producing a set of guidelines for an acceptable scheduler from the point of view of the semantics.

2.3. A Taxonomy of Fairness Notions

We shall now present some of the most widely known fairness notions.

Unconditional Fairness

Unconditional fairness (also called *impartiality*) has originally been defined for processes [LPS81]. It can be summarized as follows:

Every process proceeds infinitely often,

that is every unconditionally fair computation must admit an infinite number of occurrences of actions of each process. We have already considered an example of unconditional fairness in Section 2.2 (the set of concurrency fair computations of process P is exactly that of unconditionally process fair computations).

Unconditional fairness may be too restrictive when distributed process termination is allowed, for example see the program below:

A = **do** true → print('a') **od**
B = print('b'); stop
P = A || B

Process B, unlike process A, terminates having executed its only action. It is, therefore, unreasonable to expect process B to proceed infinitely often. The set of intuitively admissible computations should include only those computations that generate an infinite number of a's interleaved with one b (or, in other words, a^*ba^ω). However, this computation is not unconditionally fair because b does not occur in it infinitely often!

Thus, unconditional fairness should only be used in the context of non-terminating processes. Likewise, processes that have not yet terminated but have become disabled, for example by waiting to synchronise with another process, will not be correctly handled by unconditional fairness.

Unconditional fairness may also be defined with respect to non-deterministic choice [Fra86] and transitions [LPS81].

Process liveness [ApO84] is a modification of unconditional fairness to allow for the fact that processes may terminate. It can be expressed as follows:

> *Unless a process has terminated, it will proceed infinitely often.*

Now, in the program P, process liveness will admit only the intuitively acceptable computations a^*ba^ω.

However, excluding distributed termination does not reduce the expressive power of some models, for example CSP [ApF84], so considering unconditional fairness will often be sufficient.

Process liveness should not be confused with liveness as "something good will happen" [Lam77], which is a class of more general properties.

However, process liveness is still a very crude property. Note that we can re-phrase the above definition to state that *if in a computation a process has proceeded finitely often then it must have terminated.* If there are other reasons for which a process proceeds only finitely often, then process liveness is not applicable. Unfortunately, this is the case with systems that allow synchronisation; it is possible for a process to become (temporarily or permanently) disabled because it is waiting to synchronise with another process that refuses to do so.

Unconditional fairness is known in formal languages as *fairmerge*, which was introduced in [Par80] [Par81] for ω-regular languages, that is extensions of regular languages by means of an ω-iteration operator, onto languages of finite or infinite sequences over a given alphabet. Fairmerge of two such languages A, B is a language A || B that contains those sequences only, which are interleavings of pairs of possibly infinite sequences from A and B in such a way that all of any infinite sequence is absorbed.

Fairmerge is a fairness notion that was introduced as an abstraction of concurrency fairness. It applies to non-communicating, non-synchronising, concurrency, that is systems that consist of concurrent processes whose actions are totally independent. Some aspects of fairmerge have been considered for process algebras in [Hen87]. Fairmerge is not adequate to express synchronisation fairness, but it can be suitably extended.

Weak Fairness

Weak process fairness (also called *justice*) [LPS81] [CoS87] is a property that takes into account the fact that processes may become disabled (or blocked). As usual, we say a process becomes possible (or *enabled*) if some of its actions is enabled, and *disabled* otherwise. Weak process fairness can now be summarized as follows:

> *If a process is enabled continuously from some point then it eventually proceeds.*

Alternative ways of re-phrasing "continuously from some point on" [Pnu86] are: *almost always* [CoS87] (that is always after a finite number of steps) and *permanently* [Pnu86]. This definition excludes all, and only those, computations in which a process is enabled continuously and never proceeds after some point in time. Note that it follows that if some process is actually enabled continuously from some point on then it proceeds infinitely often.

Since termination is a reason for a process not to become continuously enabled, weak process fairness implies process liveness (that is, if a computation is weakly process fair, it satisfies process liveness). If a process may become disabled only when it terminates, weak process fairness coincides with process liveness. If processes are continuously enabled and never terminate, weak process fairness, process liveness and unconditional fairness coincide.

Let us consider the following example:

A = B!0 → stop

B = (x := 1;
do A?x → (print(x); stop)
[] x > 0 → print(x)
od)

P = A || B

Process A is continuously enabled to synchronise, whereas process B can repeatedly choose between synchronisation with A and the internal action. P terminates only if A and B synchronise (the second guard then becomes false). If we assume weak process fairness, then process A eventually proceeds, hence P terminates under weak process

fairness. Every weakly process fair computation leads to a finite number of 1's followed by a single 0 (that is a sequence of the form 1^*0).

Considering weak (process) fairness as a separate notion is sometimes motivated by the costs of a possible implementation [Pnu86]: it is automatically guaranteed on a truly concurrent system, whereas for multi-programming concurrency, a simple round-robin scheduler will enforce it. A "busy waiting" implementation of a semaphore is weakly fair. Stronger fairness notions require queueing mechanisms.

Several other weak fairness properties have also been introduced. One example is *weak fairness of choice of non-deterministic guards*, which may be phrased as *any guard that evaluates to true from some point on will eventually be chosen.* Let us consider the following program:

```
Q =   (x := 1;
      do    x > 0  → x := 0
      []    x > 0  → x := x+1
      od)
```

The above program terminates only if the first guard, which is continuously enabled, is ever chosen (both guards then evaluate to false). Thus Q terminates under weak fairness of choice of guards. Weak fairness of choice of guards is independent of weak process fairness.

Weak process fairness for CCS is considered in [CoS87]. A discussion of weak process, channel communication and process communication fairness for CSP can be found in [KdR83]. For an overview of weak fairness of choice consult [Fra86]. Weak fairness can also be defined at the level of events [Plo82], transitions [QuS83], and states [QuS83].

Strong Fairness

It may not always be desirable to satisfy the assumption of weak fairness, which requires that once a process has become enabled, it may not become disabled, even temporarily, if it is to proceed. We therefore relax the assumption of a process becoming enabled continuously from some point on to becoming enabled *infinitely often.* We thus arrive at a notion of *strong process fairness* (also called *fairness* [LPS81]), which can be paraphrased as follows:

If a process is enabled infinitely often then it proceeds infinitely often.

This definition disallows all, and only those, computations in which a process is enabled infinitely often but proceeds only a finite number of times. Note that strong process fairness implies weak process fairness (that is, if a computation is strongly process fair then it is weakly process fair).

Obviously, if a process never becomes enabled once it has become disabled, strong process fairness coincides with weak process fairness.

Let us consider the following example:

```
A =   B!0 → stop

B =   (x := 1;
       do    (x mod 2 = 1) and A?x     → (print(x); stop)
       []    x > 0                     → (print(x); x := x+1)
       od)

P = A || B
```

Process A is continuously ready to synchronise with B, but B can accept the synchronisation only at alternate steps (in fact, every time x contains an odd number). Thus process A is not enabled continuously, but it is enabled infinitely often. P terminates only if A and B synchronise. An infinite computation allowing only process B to proceed is weakly process fair. On the other hand, this computation is not admissible under strong process fairness, hence P terminates under strong process fairness.

Strong fairness is usually strictly stronger than the corresponding weak fairness notion. Although weak fairness is not sufficient in most cases, the distinction is again motivated by implementation considerations. In order to implement strong fairness, queues of pending requests or priority systems are needed. An example of a strongly fair semaphore is an implementation with FIFO scheduling policy.

It is also possible to define *strong fairness of choice of non-deterministic guards*, which may be described as *any guard that evaluates to true infinitely often will be chosen infinitely often*. As an example, let us consider:

```
Q =   (x := 1;
       do    x mod 2 = 1   → x := 0
       []    x > 0         → x := x+1
       od)
```

Note that Q terminates under strong fairness of choice of guards, but not under weak fairness of choice of guards (the first guard is not continuously enabled, but it is enabled infinitely often). Strong fairness of choice of guards is independent of strong process fairness.

Strong process fairness for CCS is considered in [CoS87]. For strong process, channel communication and process communication fairness for CSP consult [KdR83]. An overview of strong fairness of choice can be found in [Fra86]. Strong fairness may also be defined at the level of events [Plo82], transitions [QuS83], and states [QuS83].

Generalizations of Fairness

Equifairness

The notions of fairness introduced so far were concerned with the *independent progress* of system components. It is possible to strengthen weak, strong and unconditional fairness by taking into account the *relative* number of times the components proceed. Thus, one may arrive at a notion of *equifairness* [GFK86] [Fra86], the motivation for which is to give each guard in a group of jointly enabled guards an equal chance to proceed. Strong equifairness can be summarized as follows:

> *If a group of guards is enabled infinitely often, then there exist infinitely many time instants where all members of the group have been chosen the same number of times.*

Weak and unconditional equifairness are correspondingly defined. Let us consider the following example (adapted from [Fra86]):

```
Q =   (x, y := 0; z := 1;
       do    z > 0          → x := x + 1
       []    z > 0          → y := y + 1
       []    z > 0 ∧ x = y  → z := z - 1
       od)
```

This program does not terminate under the assumption of strong fairness of choice of non-deterministic guards: the computation taking the first guard and then alternating the first and the second guard is infinite (the third guard is only enabled in the initial state). However, it does terminate under strong equifairness because then x eventually becomes equal to y, and Q terminates since the guards no longer evaluate to true.

Strong equifairness is strictly stronger than strong fairness of choice of non-deterministic guards. (The same holds for weak and unconditional equifairness). Equifairness is usually applied to non-deterministic guards, although it seems feasible to apply it in other situations as well, for example process synchronisation.

Probabilistic fairness

Fairness has also been defined as a probabilistic property [Pnu83] [PnZ86]. We assume a computational model based on a state-transition system with probabilities attached to transitions. A *determinate* transition (i.e. non-probabilistic) is a simple edge that connects two states; it has the probability 1. A *probabilistic transition* τ is a split edge: it is an edge split into k edges with probability α_i attached, where each τ^i is called a *mode* of the transition τ. It is required that $\alpha_1 + \alpha_i + \ldots + \alpha_k = 1$. A *state predicate* is a predicate whose truth values are solely determined by the state, and a state formula is built from those by means of a first-order predicate calculus. A ϕ-state is a state that satisfies formula ϕ.

Then *extreme fairness* can be defined as follows:

> *For any state formula ϕ, and a probabilistic transition τ such that τ is taken infinitely many times from ϕ-states, each mode τ^i of τ is also taken infinitely many times from ϕ-states.*

It has been shown in [Pnu83] that in order to prove that a temporal property ψ holds with probability 1 on all computations, it is sufficient to prove that it holds on all extremely fair computations. This formalism has been used in [PnZ86] to verify multi-process protocols.

Other notions

Many more extensions or generalizations of fairness have been introduced. *Generalized fairness* [Fra86] is an attempt to abstract notions like equifairness and fairness of choice into a class of fairness notions determined by sets of pairs of state predicates. *Fair reachability of predicates* introduced in [QuS83] as a replacement for state and transition fairness is a similar notion; a computation is considered unfair if there is a predicate over states which is reachable infinitely often but states satisfying this predicate are taken only finitely often. *Relativized fairness* [QuS83] is a further refinement of fair reachability of predicates. *Conspiracy* is where a number of processes monopolize a resource, effectively preventing some process from proceeding at all [Bes84a] [Bes84b]. *Hyper-fairness* [AFG88] is a conspiracy-resistant notion adequate for multi-party operations (the reason for this distinction is that some fairness notions allow only pairwise communication; multi-party communication can be found in some models, e.g. TCSP [BHR84], and programming languages like Ada).

Related Notions

Several notions have been introduced in the past, which can be in some way related to fairness; in fact, in many cases, we believe they are fairness properties.

Finite Delay

Finite delay property was originally formulated in [KaM69], where a formalism of state-transition systems as models for parallel computations was introduced. In this formalism, computations, represented by sequences of primitive steps (transitions), correspond to *admissible sequentializations* of a concurrent behaviour. A sequence of transitions is a computation if it is either finite, in which case no more transitions remain enabled, or it is infinite, in which case it is required to satisfy finite delay property. Finite delay property can informally be described as (we shall omit the formal definition):

> *If some transition is permanently enabled from some point on then it is eventually taken.*

We believe the intention of this definition was to impose finite delay for independent, that is concurrent, actions, rather than non-deterministic choices. In fact, in [KaM69] a subclass of systems (determinate systems) that are concurrent but disallow non-determinism has been characterized; for such systems, finite delay corresponds to concurrency fairness. When adapting this property to the more general class of non-determinate systems, it seems necessary to distinguish whether a transition remains enabled in the context of concurrency or non-determinism; otherwise, the resulting property would be too strong, as it would impose concurrency fairness together with fairness of choice.

Finite delay property is a fairness property reflecting the minimum constraint on sequences of transitions/states to be admissible as sequentializations of a concurrent behaviour, and it is too weak to model synchronisation fairness. Finite delay only excludes certain infinite computations, hence, as such, it affects properties like termination and equivalence, but it bears no relation on partial correctness. It is present, as maximality of computations, in many non-interleaving models for concurrency based on causality, e.g. [LSB79] [Shi85a] [Shi88c] [MOP88] [Kwi88a]. The interleaving models, like CCS [Mil80] or TCSP [BHR84], have often been criticised for not having finite delay. A *finite delay operator* has been proposed in [Hen83], but it is not clear how this approach relates to that of [KaM69].

Livelocks

The term *livelock* is due to [Ash75], where the Floyd assertion method has been extended onto parallel programs. Ashcroft observed that, although, using induction, he can prove systems partially correct, his method could not be claimed to guarantee correctness with respect to properties like "guaranteed response", and that finite delay property does not provide a satisfactory solution. Using an example of a situation in an airline reservation system, somewhat similar to the example used in the introduction, he shows that certain events in the system can be permanently blocked by a "continually changing pattern of constraints". This is named *livelock*, as opposed to deadlock, because the system is not stopped.

Livelocks should be considered a fairness property, and also a liveness property in the sense of [Lam77] (the "good thing" would be the customer making progress). Livelocks have been examined in a Petri net setting in [Kwo79].

Unbounded Non-determinism

Fairness is often discussed in the context of unbounded non-determinism, which was first introduced in [Dij76]. This phenomenon arises in the context of guarded commands, see the program below:

```
P =   x := 0;
      do    x ≥ 0      → x := x + 1
      []    x ≥ 0      → (print(x); stop)
      od
```

Under the assumption of fairness, P would always terminate, and yet produce any, that is unbounded, natural number. This seemed in contradiction with the intuition about programming, in the sense that no feasible implementation of the above could exist.

Unbounded non-determinism has been re-considered in [Par80], where it has been observed that the above contradiction could be explained on the grounds of the existence of two interpretations of non-determinism: *loose* non-determinism and *tight* non-determinism. In tight non-determinism, non-determinism in the language describes more than one result, but a possible implementation does not have to produce all the results, only must guarantee that the results produced are described by the semantics. In loose non-determinism, a possible implementation may or may not produce more than one result; the only constraint is that every result produced is one of those prescribed by the semantics. The usual interpretation of the non-determinism of a scheduler of concurrent operations is a loose one.

Unbounded non-determinism arises in the context of infinitely branching transition systems; in [Ros88], infinitely branching systems are considered as a model for TCSP [BHR84].

2.4. Existing Formalisms and Fairness

Temporal Logic

Fairness properties are expressed most elegantly in temporal logic. There is a variety of temporal logics used to analyse program properties, which depend on the representation of the behaviour of the program in terms of its runs (i.e. sequences of states). The two main approaches here are: linear, which groups runs into a set, and branching, represented as a computation tree.

As an example, let us consider linear temporal logic [MaP81] [GPS80] [Pnu86], which is interpreted over (finite or infinite) sequences of program states. The exposition here is based on [Pnu86]. A *state formula* is any well-formed first order formula; its truth value at instant i in a sequence s is found by evaluating it on s_i. A *temporal formula* is constructed from state formulae, to which the following (basic) operators are applied, and interpreted in state s_i of some sequence s as follows:

X strong *next instant* operator (Xp is true at i if s_{i+1} exists and p is true at i+1),

U *until* operator (pUq is true at i if there exists j > i s.t. q is true at j and for all k, i<k<j, p is true at k),

P strong *previous instant* operator (Pp is true at i if i>0 and p is true at i-1),

S strong *since* operator (pSq is true at i if for some j, 0<j<i, q is true at j and for all k, j<k<i, p is true at k),

Some of the derived operators are:

F *sometime* in the future (Fp = **true**Up), often denoted as diamond,

G *always* in the future (Gp = ¬F ¬p), often denoted as box.

Examples of temporal formulae are:

$p \rightarrow Fq$	if p now then eventually q
$G(p \rightarrow Fq)$	every p is followed by a q
$F(Gp)$	eventually permanently p
GFp	infinitely often p (for infinite sequences only)
$GXFp$	infinitely often p (for finite and infinite sequences)

A formula p is said to be *satisfiable* if there exists a sequence s and a position j such that p holds at j (denoted $s_j \models p$). A formula p is *valid* if for all sequences and positions $j < len(s)$, $s_j \models p$.

A formula p is *valid over program* R if for every sequence (computation of R) s and $j < len(s)$, $s_j \models p$.

Temporal logic distinguishes between safety and liveness properties. A safety property is characterized (up to initial equivalence) as:

$$Gp$$

where p is some past-formula, that is p holds over all finite prefixes. Any formula constructed out of past formulae, the logical operators $\wedge$, $\vee$, and the future temporal operators G, U is a safety property.

A (basic) liveness property complements a safety property by requiring that certain finite prefix properties hold at least once, infinitely many times, or continuously from some point on, that is correspondingly:

$$Fp,\ GFp,\ FGp$$

for some past-formula p.

The following are examples of fairness properties in temporal logic, assuming p stands for "process enabled" and q stands for "process taken":

$Gp \rightarrow Fq$, i.e. $F(\neg p \vee q)$	weak fairness
$GFp \rightarrow Fq$, i.e. $F(G(\neg p) \vee q)$	strong fairness.

Formally, we have introduced a language L(X,U,P,S), also called TL (linear temporal logic). Its subclasses TLF and TLP correspond to future and past temporal logics L(X,U) and L(P,S), which have equivalent expressive power with TL. It is known that L(U) has

the same expressive power as L(F,X,U), but L(X,F) is not expressively complete [GPS80] - for example, fairness properties shown above could not be expressed without Until. Another point is that linear temporal logic, being a non-counting formalism, does not have the same expressive power as regular expressions.

Linear temporal logic provides temporal operators that describe events along a single computation path. In branching-time logic, the temporal operators quantify over the paths that are possible from a given state. It is an interesting question to compare the expressive power of linear and branching time temporal logic. A temporal logic language CTL*, which combines both linear-time and branching-time operators, is introduced in [EmH86], thus making it possible to obtain the results of such comparison. It is shown that CTL* strictly subsumes B(L(F,X,U)). Branching time logic B(L(F)) has different expressive power than its linear counterpart L(F).

Automata and ω-regular languages

In order for fairness properties to be included in the analysis of system behaviours, it is necessary to allow both finite and infinite sequences. This implies extending formal languages to infinitary languages, which, in turn, requires re-considering finite state automata as acceptors of languages.

Infinitary languages have been investigated by a number of authors, e.g. [BoN79] [Par81]. *ω-regular languages* are a natural extension of regular languages with an ω-iteration operator, that is iteration an infinite number of times. *ω-automata* [Par81] are formed from standard finite automata by the addition of structure for accepting infinite sequences. Two (equivalent in the non-deterministic case) varieties are: *B-automata*, due to Buchi, with an additional set of green states which must be visited infinitely often, and *M-automata*, first introduced by Muller, where a set of accepting states with a similar requirement is specified. The class of ω-regular languages is recognised by B-automata.

The relationship between ω-regular languages and a concurrency operator (fairmerge) is dealt with in [Par81]. Fairmerge of infinitary languages is a function that interleaves pairs of sequences in such a way that the whole of an infinite sequence is taken. The class of ω-regular languages is shown to be closed under fairmerge. Fairmerge corresponds to concurrency fairness for non-communicating concurrency, but it may be extended onto a communication merge [Par85] [BeK86].

Fairness in (labelled) finite state automata is discussed in [PRW87]. Only the case of strong fairness is considered there. The notions of *edge*- and *letter*-fairness (that is transition fairness and fairness with respect to transition labels) are distinguished. These notions are generalized onto (finite) *paths* (i.e. sequences of edges) and *words* (i.e. sequences of letters), and the hierarchy of languages fair with respect to a given notion is discussed. The relationship with Buchi and Muller automata is also considered.

Process Algebras

Process algebras are algebraic languages for the specification of concurrent processes and the formulation of properties of such processes. They are usually introduced together with the rules of algebraic calculus.

CCS [Mil80] is a calculus whose closed expressions correspond to processes. The behaviour of processes is determined by the (transition) rules of the calculus. The language allows: prefixing a process E with an action (aE), non-determinism (+), concurrency (|), recursion (fix), and restriction (\). Synchronisation is enforced only under restriction; otherwise processes may choose to proceed autonomously or synchronise. The rules of CCS (with minor changes) have been included in the Appendix. The following is an example of a CCS process:

(E | F)\b

where E = fix X.(aX + bNIL), F = $\overline{b}$NIL. This process terminates only if E and F eventually synchronise.

Fairness in (pure) CCS without restriction has been examined in [CoS84], where it is defined as an issue to do with concurrent processes. No distinction between weak and strong fairness was needed there because processes never become locally disabled while waiting to synchronise with other processes. Pure CCS with restriction has been investigated in [CoS87], where weak process fairness is defined in the sense that no process that is almost always enabled can be delayed indefinitely, and strong fairness is defined correspondingly.

The main concern of [CoS87] is to provide a positive treatment of fairness, i.e. generate only the derivations that are fair. The calculus of CCS has been extended to a labelled calculus by adding labels identifying the path to a component of an expression. These

labels have then been used to uniquely identify autonomous actions and processes in a given expression. A subclass of live processes, that is processes that became active and have not made a move yet, is distinguished.

In this setting, the set of rules for Weak Fair CCS and Strong Fair CCS have been developed. Both sets of rules are finite and do not involve random assignment. Weak Fair CCS is based on a local characterization of admissibility. On the other hand, Strong Fair CCS can not be similarly characterized and involves predictive choice [Mil80], that is one needs to base the definition on an infinite tail of a derivation in question.

A metric characterization of weak and strong fairness in CCS was presented in [Cos84]. The conclusion there is that fair infinite derivations of a given expression can be characterized as limits of infinite chains of finite derivations. Although a generalization of the results onto the class of all expressions is proposed, it does not seem abstract enough as it is based on pairs consisting of an expression and its derivation. Fairmerge was discussed in a CCS setting in [Hen87].

CSP [BHR84] [Hoa84] (also known as TCSP) is a process algebra, which distinguishes between internal and external choice and allows hiding (abstraction). Inter-process communication is based on (n-communicating) joint actions, rather than binary synchronisation in the sense of CCS. The usual semantic model for CSP is the failures model, which is a transition system extended with additional failure and ready sets to model the interaction of processes with their environment. Fairness is not expressible within CSP [Hoa84]. In [Ros88], extending the failures semantics is proposed so that unbounded non-determinism is allowed. It is not known if fairness issues have been considered in this setting.

ACP [BeK86], the Algebra of Communicating Processes, is not tied to a particular model. It is based on bisimulation semantics [Par81]. The primitives include, apart from the usual ones, a left merge operator, in terms of which a parallel composition is defined. Communications are generalized to multisets of actions that can happen simultaneously. ACP is extended with abstraction and encapsulation operator in [vGl86]. A process is fair if for any improbable path (that is containing infinitely many exits) the probability that it will be executed is zero. Fair abstraction rule, that states that any invisible infinite path may be discarded, is shown to be satisfied in this algebra. It is not clear how this approach relates to the rest of the research.

Infinitary algebra is discussed in [Par85]. It is influenced by CCS, restricted to regular behaviours (that is corresponding to finite-state machines). Fairness properties are introduced as operators, which define restrictions that affect only infinite behaviours. This is achieved by introducing an infinitary agent A<<L>>, where A is an agent in the sense of CCS and L is an ω-regular language containing no finite sequences. A<<L>> behaves as the agent L, except that its infinite behaviours must be members of L. Equivalences then can be proved under such constraints. This process algebra is applied to protocol verification.

Net Theory

A number of papers have been published on fairness in Petri nets. Net theory is founded upon the idea of a Petri net, which is commonly recognised as one of the first models for concurrency based on *causality*, rather than interleaving. A *Petri net* [RoT86] is a triple N = (S, T, F), where S is the set of *places*, T is the set of *transitions*, and $F \subseteq S \times T \cup T \times S$ is the *flow relation*. A *marking* is a function M: S → N that defines the number of tokens in a given place. A transition is *enabled* (may fire) if all its input places contain at least one token. As a result of a transition firing, one token is added to each output place. The behaviour of a net is often represented as an alternating sequence of markings and transitions: $M_0t_1M_2t_2$... Petri nets can be viewed as transition systems, with markings forming the states.

Transition fairness has been investigated in [Bes84a] [Bes84b], where an infinite hierarchy of notions of fairness, motivated by the need to exclude conspiracy, has been introduced. This hierarchy collapses to a single notion for a simpler class of nets (confusion-free). Another approach is that of [CaV84], where notions of fairness for three granularity levels are defined in terms of subclasses of infinitary languages: markings, (bounded) places, and transitions. Transition fairness corresponds to fairness of choice of transitions, and marking fairness to fair reachability of states. The hierarchies of classes of fair languages are investigated in [CaV84], and some decidability questions are answered in [Car87].

It is not clear how processes should be defined for Petri nets; a variety of decompositions are possible. In [Mer86], a notion of a process based on occurrence nets has been introduced. Transition fairness and marking fairness, proved to be independent notions, have also been discussed there. The hierarchy of marking fairness is shown to collapse.

In [Kwo79], livelocks have been investigated in parallel programs modelled by a Petri net, and a technique for verifying the absence of livelocks has been introduced.

Transition Systems

Transition systems, with a countable set of states, seem to be the underlying structure of all models for discrete systems. A number of fairness notions related directly to transition systems as models for non-deterministic or concurrent programs have been introduced. Most commonly, they correspond to fairness at the granularity level of transitions or states, although some generalizations have also been proposed. Transition fairness is too general as a fairness notion, because it depends on what constitutes a state and what a transition represents.

In [QuS83], labelled transition systems are used as models for non-deterministic programs. A state is a vector of program variables, while transitions correspond to guarded commands and are enabled in a given state if the corresponding guard evaluates to true. In this formalism, fairness with respect to transitions, fair choice of states, and fair reachability of predicates over states (all in their strong form only) are criticised for a number of anomalies, including not being preserved under syntactical transformations. Relativized fairness is introduced instead and a branching time temporal logic is formalized to prove properties under fairness assumptions.

Obviously, transition fairness as defined in [QuS83] corresponds to fairness of choice. However, when concurrency is represented as non-deterministic interleaving, transition systems may be used to model parallel composition of processes [MaP81]. According to this approach, states correspond to vectors of variables together with control information about each process (that is, the location of control of each process at a particular instant in time). Here, (weak, strong or unconditional) transition fairness corresponds to (weak, strong or unconditional) process fairness. In [Pnu86], transition systems are extended to *fair transition systems* by adding restrictions on justice and fairness (i.e. weak and strong fairness).

Fairness for a concurrent while language modelled as a transition system is also examined in [Plo82].

Denotational Semantics

The issue of fairness raises a few important questions in denotational semantics [Sto77], especially in relation to continuity and the Scott hypothesis. Denotational semantics defines program constructs in terms of functions over *domains*, usually complete partial orders. It is commonly accepted that all computable functions can be expressed in terms of continuous functions over domains. It is also recognised that the *powerdomain* construction [Plo82] is needed to provide denotations of non-deterministic or concurrent programs.

In [Par80], the notion of continuity, in the sense of preserving least upper bounds of infinite chains, is extended to uncountable chains. A relational approach to denotational semantics of non-deterministic programs is proposed. The problem of (concurrency) fairness and unbounded non-determinism is re-considered. It is shown that the intuitively acceptable fixpoint solution to the fairmerge of languages 0^ω and 1^ω is neither the minimal, nor the maximal fixpoint. This suggests that generalized fixpoints may be needed to deal with fairness.

In [Plo82], a powerdomain semantics for a concurrent while language (without internal non-determinism) is discussed. Weak and strong event fairness is distinguished, and fair parallel operators $\|_n$ and $\|^n$, which put an explicit bound n on the number of moves the process on the corresponding side can make, are defined.

The question of the relationship of the operational semantics and denotational semantics (in terms of infinite streams) of a process algebra is addressed in [BMO87]. The issue of fairness is not dealt with there, but an observation is made that fairness properties correspond to behaviours that are not closed in the topological sense. The topology considered is one with respect to the metric space determined by the usual distance over streams defined as $2^{-(n+1)}$, where n is the length of the longest common prefix. A suitable example would be the language {a*b} representing computations that are fair with respect to communication b. This language is not closed, but $\{a^*b\} \cup \{a^\omega\}$, containing an unfair computation a^ω, is.

2.5. Proving Properties under Fairness Constraints

Properties of programs can be split into two classes: *safety* (invariance) and *liveness* properties. This is motivated by different proof techniques required in each case: for safety, structural induction based on invariants suffices, while liveness requires the method of well-founded sets. A number of axiomatic systems originating from Floyd assertion method have been introduced to reason about the correctness of concurrent programs, e.g. [Ash75] [OwG76] [MaP81] [Pnu86]. Each axiom corresponds to a syntactical construct in the language.

Partial correctness and mutual exclusion are examples of safety properties. Termination and guaranteed response are liveness properties. Safety properties, as opposed to liveness, are not usually affected by fairness properties.

Well-founded Sets

The main proof technique used to show termination (and, for that matter, liveness properties) of programs is based on well-founded sets. A *well-founded set* (A, <) is a partially ordered set in which there does not exist an infinite strictly decreasing sequence. Elements of such a set may be used to define a *ranking function* $\rho: S \rightarrow A$ [LPS81], where S denotes the set of states of a given program. The ranking function should be defined so that, as the computation proceeds through state transitions, the value of the ranking function for the visited states decreases. Since no infinite ranking sequence is possible, the program must terminate. The program terminates if, and only if, such a ranking function exists.

For deterministic programs, it is sufficient to require that the ranking function decreases at every step. However, when concurrent, or non-deterministic, programs are considered, this requirement may be too strong. Often the ranking function could be defined only for the fair sequences. Let us see the following example:

A = (b := false)
B = **do** true → skip **od**

P = (b := true; A || B)

As long as only process B is taken, nothing changes in the state, so no ranking sequence exists. However, assuming (weak process) fairness, the program terminates.

The solution suggested in [LPS81] is to define ranking sequences, which only decrease *eventually*. Thus, we distinguish between a *helpful* direction, that is one that decreases the ranking function (b := false), and an *indifferent* one, that does *not* increase it. Now, using a proof rule which incorporates the required fairness constraint, we can show that, as the computation proceeds, the ranking function *does not increase* and, by the fairness constraint, the helpful direction is eventually chosen. Thus for program P, assuming weak process fairness, the well-founded set would be {0, 1}, with ρ(true) = 1, ρ(false) = 0, as the states could be identified with the value of b.

A complication of this method, when used to verify concurrent or non-deterministic programs, is that arbitrary (not just countable) well-founded sets are required. A ranking sequence would typically include *transfinite ordinals*, rather than just natural numbers, hence *transfinite induction* may also be required. Ordinal numbers are an extension of the concept of natural numbers beyond ω, and are defined as follows: $0 = \varnothing$, $n + 1 = n \cup \{n\}$. From now we proceed by taking $\omega = \{0, 1, \ldots\}$, $\omega + 1 = \omega \cup \{\omega\}$, and so on [Hal60]. Ordinals are strictly ordered by set membership and form a well-founded class.

Let us consider the following example:

```
A =   x := 0; comm := true;
      do    comm        → x := x + 1
      []    comm        → (B!x; comm := false)
      od

B = (A?y; C!y; stop)
C = (B?z; do z > 0 → z := z - 1 od)

P = ( A || B || C)
```

Clearly, C is terminating so, as long as A || B terminates, P will terminate. Note that A || B always terminates under the assumption of weak process fairness. Process A produces an unbounded natural number, which is then sent to C, who decrements it until it reaches zero. The well-founded set in this case is ω, the second guard in A is helpful and the first one indifferent. The ranking function is defined ρ(true) = ω, ρ(false) = x, where the truth values in this definition correspond to the value of the variable comm, which we have included to indicate that communication between A and B has occurred. (Note that taking a

bounded value here was impossible because indifferent directions must not increase the ranking function.)

The well-founded sets method also applies to liveness properties [Pnu86], rather than just termination. In this approach, fairness is incorporated as a constraint into proof rules, that is a variety of proof rules are formulated depending on the actual notion of fairness.

A compositional approach is also possible, which allows for modules to be specified and verified independently of their environment. This is achieved by the introduction of a notion of an interface. Compositional approach to temporal logic is discussed in [BKP84]. It is not clear how fairness constraints are dealt with there.

Well-founded sets are also used to prove termination of non-deterministic programs [Fra86]. Two inter-reducible methods seem to be used in this case: the *state-directed choice* (essentially the ranking function with helpful directions as described above), and the *ordinal-directed choice* (based on the existence of a parametrized invariant). A variety of proof rules, one for each fairness notion, are considered in [Fra86] together with their soundness and semantic completeness. Termination proofs are considered sufficient since other liveness properties can be reduced to termination [GFM81].

Transformational Approach

An alternative method for tackling termination of non-deterministic programs is the method of the *explicit scheduler* [Fra86]. It is based on the transformation of the original program into a derived program that uses random assignment. This way, it is possible to find a ranking sequence in the derived program which decreases at every computation step. Thus, a simpler rule could be used to reason about the termination of the original program. *Random assignment* is a statement of the form x := ?, where ? stands for any natural number.

The method of explicit scheduler is analysed in [Fra86], where soundness and completeness are also considered.

Positive Approach

The methods we have discussed so far were essentially negative approaches based on two-level semantics. At the lower level, all computations, whether fair or unfair, were allowed. The proof rules incorporated a variety of fairness constraints, thus making it possible for the proof to reason only about computations that are fair. Computations that are not admissible under a given scheduling policy were considered irrelevant.

The positive approach is concerned with generating the fair computations only. A positive approach to fairness in CCS has been considered in [CoS87]. This was achieved by introducing two sets of rules, namely Weak Fair CCS and Strong Fair CCS, which are mathematically more complex than the standard rules for CCS. Any property verified by means of those rules automatically holds for all, and only, fair computations. Such an approach may be criticised for being inflexible, as any other notion of fairness would require a new set of rules.

Automatic Verification

The obvious disadvantages of manual verification of program properties may be overcome by a mechanization of the verification process. In temporal logic, so called *model checking* [ClG87] provides such facilities. This method relies on the representation of finite-state programs as labelled state-transition graphs (called Kripke structures). The algorithm developed for CTL [ClG87] which determines if a formula f_0 is true in the state s has complexity $O(len(f_0)*(|S|+|R|))$, where S denotes the set of states and R is the reachability relation.

Incorporating fairness constraints involves extending the state-transition graph with a set of fairness predicates, each of which is required to hold infinitely often along a computation path. The temporal operators now quantify over fair computation paths. The complexity of the model checking in this case increases by the factor |F|, where F denotes the set of fairness predicates. A potential problem of this method is the exponential state explosion.

Other model checking systems that allow for fairness include [CaH87] [EmL87].

2.6. Applications

The techniques developed for program verification under fairness constraints have been used, in isolated cases, in some practical applications. The work understandably concentrates on mutual exclusion algorithms and protocol verification, as fairness properties become important when modelling synchronisation or data transfer over a faulty medium.

Concurrent Algorithms

When designing and implementing mutual exclusion algorithms, some reasoning about fairness is needed to ensure that a process requesting access to the critical section will eventually be allowed to enter. Mutual exclusion algorithms are often verified in the literature to exemplify the proof technique considered. Peterson's algorithm is shown correct in [MaP81] [Pnu86] (the proof is based on linear temporal logic). Other examples are [OwL82] proved using the "proof lattice" technique. The same algorithm is automatically verified in [ClG87] by the model checking algorithm for the CTL temporal logic. In [PnZ86], a version of n-process mutual exclusion is verified using the notion of extreme fairness (that is probabilistic fairness). The algorithm is based on symmetric protocols that do not share a writeable storage, but other processes are allowed to read the value of the private variable of a given processes. Algorithms with the same constraint, deemed necessary in distributed systems, are analysed in [Lam86]. Apart from fairness, some failure-tolerance properties are also considered there.

Fairness of synchronisation primitives has been characterized in [Mar81]. Unbounded communication primitives and two versions of semaphore operations are axiomatized. In [KaL76], monitors that guarantee fairness are implemented as an extension of Pascal with concurrency.

Protocol Verification

The verification of network communication protocols is of increasing importance. The alternating bit protocol is verified under fairness constraints using many techniques, e.g. a

manual proof in terms of an infinitary process algebra [Par85], and an automatic verification in CTL [Cla86]. The CSMA/CD protocol is verified in [Par85] also using an infinitary process algebra.

Verification of multi-process probabilistic protocols is discussed in [PnZ86], where the notion of extreme fairness is employed. The protocol used there is based on n-process mutual-exclusion algorithm assuming no writeable storage for communication.

Automatic verification using a model checking system written in PROLOG has also been discussed in [CaH87]. For more detailed information on the subject of protocol verification consult [Par85].

Programming Languages

It is an interesting question whether fairness constraints formulated for existing programming languages, at the stage of the language design or implementation, are adequate. In [AFK87], criteria for appraisal of fairness notions have been introduced. These are: *feasibility*, that is whether some computation of a program remains having excluded all computations that are unfair with respect to the given notion, *equivalence robustness*, that is if fairness respects the equivalence induced by the model, and *liveness enhancement*, that requires that additional liveness properties hold for some program. When fairness notions for CSP [Hoa78] have been analysed, only strong process fairness is shown to satisfy all three criteria. A number of fairness notions for communication in Ada has also been considered.

Other work concerning programming languages revolves around synchronisation primitives, for example semaphores [Mar81] or monitors [KaL76].

2.7. Conclusion

Fairness and fairness-related notions stem from the observation that a certain undesirable phenomenon, often present in infinite computations admissible under a given semantics, and almost always relating to the lack of progress of some component of a system, must be

disallowed. When first introduced, for example in [KaM69] [Ash75] [LPS81] [Par81], fairness was defined as an issue to do with concurrency. The motivation for the introduction of fairness was to make the *admissible* behaviour realistic from the point of view of a concurrent implementation on a multi-processor system. The definitions of fairness were often elusive, informal, and heavily dependent on the particular model used. This situation has led to the emergence of the multiplicity of model-specific fairness notions, guided by concerns other than concurrency, for example fairness of choice; as a result, the originally intended intuition of fairness has been obscured.

It should be recognised that fairness is not a monolithic notion; rather, in its present form, it is a collection of (mostly independent) properties, which are relative to the choice of the granularity level and the strength required. Nevertheless, fairness properties exhibit certain features that distinguish them from other properties. All fairness notions known to the author exclude some infinite behaviours while all finite behaviours are considered fair. Also, fairness usually requires that some system component makes progress *infinitely often*, without putting an explicit bound on the delay (the only constraint is that this delay is *finite*). Notions like *infinitely often*, *component becomes possible*, *progress has been made* need to be formalized in order to express fairness.

It has been maintained that fairness has no effect on partial correctness and, in general, safety properties. It does, however, affect liveness properties, for example program termination. This observation relies on the fact that, when some infinite computations have been excluded, it is often the case that no infinite computations are admissible, hence the program will be terminating. Fairness is introduced in most formalisms as a *constraint* [Pnu86], that is an assumption incorporated at the verification stage, for example as a premise of a proof rule, which is used to prove properties other than fairness. On the other hand, fairness may be treated as a *property* of programs, that is it may be added to the list of properties that form a specification for that program. Given notations expressive enough, statements about fairness of a given program can be made and verified, although this may have some effect on the complexity of the proof [ClG87].

It became apparent that some existing formalisms could not adequately incorporate fairness as a property expressible within the formalism. Many models had to be extended with infinite computations. In temporal logic, it has been observed that some properties could not be expressed without certain powerful temporal operators [GPS80] [EmH87]. In verification techniques, incorporating fairness has called for transfinite induction [LPS81] [Fra86]. In denotational semantics, continuity had to be re-considered [Par81].

Further difficulty arises when different behavioural structures are used as models, for example, when non-deterministic interleaving is used to represent concurrency. At this high level of abstraction, no restrictions are usually imposed on admissible sequentializations of concurrent computations (for example, as a finite delay property [KaM69]). For such approaches, fairness with respect to concurrent processes may be reduced to fairness of choice for a particular scheduler. When transferring fairness onto other behavioural structures, for example those based on causality, certain anomalies could come to view [Kwi88b].

Due to a number of different guises fairness takes in different models, it is not clear at this stage how best to approach fairness. Doubts have often been expressed if powerful formalisms like transfinite induction are really needed to represent a property so easily, on the face of it, implementable in practice. Also, some quantitative methods, rather than qualitative that have been applied until now, would be beneficial to model bounded delays necessary in real-time systems.

Recently, some criticism was raised [Dij88] as to whether fairness is a "workable notion" since no finite experiment can be set up to prove or disprove it. We would argue against this statement and tend towards [ChM88] thinking that fairness is a useful abstraction, especially when related to concurrency, and, as such, although undetectable by finite experiments, it nevertheless is a property that can be formally established. However, further research is needed in order to provide an adequate characterization of fairness properties independent of the intricacies of the models used. This seems a worthwhile goal, considering the pleasing fact that the research in concurrency has begun to converge.

3

The Model

Finitary trace languages were introduced in [Maz77] as an attempt to generalize formal languages in order to represent concurrent behaviour. The notion central to trace theory is one of independency relation over action symbols, which describes the potential for concurrency. Independency gives rise to trace equivalence over sequences of action symbols, which is a congruence in the monoid of strings. Asynchronous transition systems [Shi85a] [Bed87] [Shi88c] are an extension of sequential labelled transition systems with an independency relation.

In this chapter, we introduce the framework within which we shall investigate the relationship of fairness and non-interleaving concurrency. First, we enhance trace theory by extending trace equivalence onto infinite strings. Then we discuss asynchronous transition systems, which we extend with a notion of process structure to determine the contribution of individual concurrent agents to the overall behaviour of the system. Asynchronous transition systems are given interleaving and non-interleaving semantics in terms of derivations and traces respectively. An alternative representation of traces based on projective equivalence and preorder in the monoid of strings derived from [Shi88c] is also developed; its product is vector semantics, which relates behaviours to concurrent agents. It turns out that projective equivalence coincides with trace equivalence.

Finally, we investigate properties of infinite traces and show that the set of all traces forms an infinitary monoid and a domain. The relationship of asynchronous transition systems and infinitary trace languages as their behaviours is formally established.

We use examples based on a variant of CCS [Mil80], the syntax and semantics of which can be found in the Appendix, and Condition/Event nets, which have been defined in Chapter 6.

3.1. Preliminaries

This section gives a summary of basic notions referred to in the thesis.

Sets, Relations and Domains

The definitions in this section are based on [ScG87] [GHK80] [Shi88c].

Let P denote a set, $R \subseteq P \times P$ denote a relation over P. R is a *preorder* iff it is reflexive and transitive. R is a *partial order* iff it is reflexive, anti-symmetric and transitive. R is an *equivalence* iff it is reflexive, symmetric and transitive. R is a *congruence* wrt a (binary) operation $\oplus$ in P iff $a \, R \, b \Rightarrow a \oplus c \, R \, b \oplus c$ and $c \oplus a \, R \, c \oplus b$ for all c.

Let S denote a set and $\oplus$ a binary operation over S. $(S, \oplus, \varepsilon)$ is a *monoid* iff ε is an identity for $\oplus$ and $\oplus$ is associative.

Let $(P, \leq)$ be a partially ordered set *(poset)*. For any subset X of P we define the *prefix closure* of X, denoted $\downarrow X$, as $\{y \in P \mid \exists x \in X: \; y \leq x\}$ (also called the *lower set*). The *suffix closure* of X, denoted $\uparrow X$, is defined by $\{y \in P \mid \exists x \in X: \; x \leq y\}$ (also called the *upper set*). $\downarrow\{x\}$ and $\uparrow\{x\}$ are abbreviated to $\downarrow x$ and $\uparrow x$.

Let $(P, \leq)$ be a poset. We say a is a *lower bound* of a set $X \subseteq P$, and b is an *upper bound*, provided that $a \leq x$ for all $x \in X$, and $x \leq b$ for all $x \in X$, respectively. If the set of upper bounds of X has a unique smallest element, we call this *least upper bound* (denoted *lub*). Similarly the *greatest lower bound* is denoted by *glb*. P is a *lattice* iff every pair of elements $x, y \in P$ has a least upper bound and a greatest lower bound. P is a *complete lattice* iff every $X \subseteq P$ has a least upper bound and a greatest lower bound.

Let $(P, \leq)$ be a poset. $x, y \in P$ are *compatible* iff the set $\{x, y\}$ possesses a least upper bound. $X \subseteq P$ is *pairwise compatible* iff every $x, y \in X$ are compatible. We say that $(P, \leq)$ is *coherent* iff every pairwise compatible subset $X \subseteq P$ has a least upper bound in P.

Let (P, $\leq$) be a poset. $X \subseteq P$ is *totally ordered*, or a *chain*, iff for every $x, y \in X$, either $x \leq y$ or $y \leq x$. $X \subseteq P$ is a *directed set* iff it is non-empty and $\forall\, x, y \in X \;\; \exists\, z \in X$ such that $x \leq z$ and $y \leq z$. Otherwise it is *branching*. Equivalently, $X \subseteq P$ is a directed set iff its every finite subset has an upper bound in X.

Let (P, $\leq$) be a poset. An element $x \in P$ is a *complete prime* iff, for every $X \subseteq P$, if $x \leq$ *lub*(X) then there exists $y \in X$ such that $x \leq y$. Let *Pr*(P) denote the set of complete primes of (P, $\leq$). (P, $\leq$) is *prime algebraic* iff for every element $x \in P$, the least upper bound of the set $\{y \in Pr(P) \mid y \leq x\}$ exists and equals x.

Let (P, $\leq$) be a poset. P is a *complete partial order* (cpo) iff

(i) P has a least element, and

(ii) if $X \subseteq P$ and X is directed, then X has a least upper bound in P.

Note that since every pairwise compatible set is directed, every coherent poset is a cpo (e.g. see [Shi88c]).

Let (D, $\leq$) be a cpo. $x \in D$ is a *finite element* if, whenever $M \subseteq D$ is directed and $x \leq$ *lub*(M), then there exists $y \in M$ such that $x \leq y$. The set of all finite elements of D is denoted B_D. D is *algebraic* iff, for every $x \in D$, the set $M = \{y \in B_D \mid y \leq x\}$ is directed and *lub*(M) = x. D is a *domain* iff D is algebraic and B_D is countable.

Given complete partial orders D, E, a function $f: D \rightarrow E$ is *monotone* iff $f(x) \leq f(y)$ whenever $x \leq y$. If f is monotone and $f(lub(M)) = lub(f(M))$ for every directed M, then f is said to be *continuous*.

The following are standard results in domain theory [ScG87]. If D, E are complete partial orders, then D×E is a cpo with coordinatewise ordering $\langle x,y\rangle \leq \langle x',y'\rangle$ iff $x \leq x'$ and $y \leq y'$. If D, E are domains, then D×E is a domain with $B_{D\times E} = B_D \times B_E$. Given a cpo F and continuous functions $f: F \rightarrow D$, $g: F \rightarrow E$, there is a continuous function $f\times g: F \rightarrow D\times E$ given by (f(x), g(x)).

Let (D, $\leq$) be a domain. For any subset X of we define the *prefix closure* of X, denoted $\downarrow X$, as $\{y \in D \mid \exists\, x \in X:\; y \leq x\}$ (the lower set). We shall also distinguish closure with respect to finite prefixes, denoted $\downarrow^{fin} X$, which is defined as $X \cup \{y \in B_D \mid \exists\, x \in X:\; y \leq x\}$. The *suffix closure* of X, denoted $\uparrow X$, is defined by $\{y \in D \mid \exists\, x \in X:\; x \leq y\}$ (the upper set). $\downarrow\{x\}$, $\uparrow\{x\}$ and $\downarrow^{fin}\{x\}$, for $x \in D$, are abbreviated to $\downarrow x$, $\uparrow x$ and $\downarrow^{fin} x$ respectively.

Infinitary Languages

Let A denote a finite alphabet; ε denotes the empty sequence, A^* represents the set of all finite sequences (also called *strings* or *words*) over the alphabet A, A^ω represents the set of all infinite sequences over A, and A^∞ is the union of A^* and A^ω. The length of a sequence $x \in A^\infty$ is denoted $|x|$ (ω for infinite sequences). For $i \in N^+$, $x \in A^\infty$, x(i) denotes the ith symbol of the sequence x if it exists, and ε otherwise, where N^+ denotes the set of natural numbers excluding zero.

We follow [BoN79] in the way we define prefix ordering and concatenation over A^∞. The *prefix order* over A^∞ will be denoted by $\leq$ and is defined below:

$$\forall x, y \in A^\infty: \; x \leq y \Leftrightarrow \forall i \in N^+: (i \leq |x| \Rightarrow x(i) = y(i))$$

It should be noted that only finite proper prefixes are distinguished (i.e. for all $x, y \in A^\omega$: $x \leq y \Rightarrow x = y$).

We define x[i] as the sequence $x(1)...x(\min\{i, |x|\})$. The set of finite prefixes of x, $\mathrm{Pref}^{\mathrm{fin}}(x)$, is $\{x[i] \mid i \in N^+\}$, whereas the set of all prefixes of x, Pref(x), is $\mathrm{Pref}^{\mathrm{fin}}(x) \cup \{x\}$.

Concatenation is extended onto A^∞ by taking as the concatenation of sequences x, y:

$\forall x \in A^*, y \in A^\omega$: the infinite sequence xy
$\forall x \in A^\omega, y \in A^\infty$: the infinite sequence x.

Concatenation will be denoted as juxtaposition. A^∞, together with the above concatenation, forms a monoid (proof of associativity is clear).

We define a binary left cancellation operator, denoted /: $A^* \times A^* \rightarrow A^*$, as follows:

$$\forall x \in A^*, y \in A^\infty: \quad y/x = z \Leftrightarrow (x \leq y \wedge xz = y).$$

Example. Concatenation: $a^\omega b = a^\omega$, $a^\omega b^\omega = a^\omega$, $a^n a^\omega = a^\omega$.
Left cancellation: $aab/a = ab$, $a^n/a^n = \varepsilon$; a^n/b is undefined.

A^∞, together with the above prefix ordering $\leq$, forms an algebraic cpo, and, since the set A^* of finite elements of A^∞ is countable, it follows that $(A^\infty, \leq)$ is a domain.

Proposition 3.1.1. $(A^\infty, \leq)$ is a domain. The set of finite elements is equal to A^*.

Proof.

(i) Obviously, ε is the least element. It follows from the definition of the prefix ordering $\leq$ that every directed subset of A^∞ is totally ordered. Suppose $X \subseteq A^\infty$ is directed. If X is finite, then the largest element of X is the lub. If X is infinite, its elements form an infinite chain:

$$x_1 \leq x_2 \leq \ldots \leq x_i \leq \ldots$$

and its least upper bound $d \in A^\omega$ can be constructed as follows:

$$d_i = \begin{cases} x_1(i) & i \leq |x_1| \\ x_j(i) & |x_j| < i \leq |x_{j+1}| \end{cases}$$

Thus we have shown that $(A^\infty, \leq)$ is a cpo.

(ii) We now show that A^* are the finite elements. Suppose $x \in A^*$ and $M \subseteq A^\infty$ is directed. Thus M must be totally ordered and lub(M) exists as shown in part (i) of this proof. We need to show the existence of $y \in M$ such that $x \leq y$ whenever $x \leq$ lub(M). Assume $x \leq$ lub(M) and M is finite, then lub(M) $\in$ M, and we can take y = lub(M). If $x \leq$ lub(M) and M infinite, then either lub(M) $\in$ M, in which case we take y = lub(M), or lub(M) $\notin$ M. In the latter case, lub(M) must be an infinite word while x is finite, and $x <$ lub(M). In case $x \in$ M, we take y = x. Otherwise, M contains words of unbounded length which are prefixes of lub(M). x is also a prefix of lub(M) and it is bounded, hence it must be possible to choose $y \in$ M such that $x \leq y$.
Clearly, elements of A^ω are not finite. As a counter-example consider any $x = A^\omega$ and any infinite directed set M such that lub(M) = x but $x \notin$ M.

(iii) We now show that $(A^\infty, \leq)$ is algebraic. Let $x \in A^\infty$ and define $M = \{y \in A^* \mid y \leq x\}$. M is clearly a total order, hence it is directed. It is easy to see that x = lub(M). Since A^* is countable, it follows $(A^\infty, \leq)$ is a domain.

❑

$(A^\infty, \leq)$ is also a prime algebraic and coherent poset (proof can be found in [Shi88c]). The set of complete primes is equal to $A^* - \{\varepsilon\}$.

A subset L of A^∞ is called an *infinitary language*. We extend the set of finite prefixes onto languages by $\text{Pref}^{\text{fin}}(L) = \{x \in \text{Pref}^{\text{fin}}(y) \mid y \in L\}$. An *adherence* of a language $L \subseteq$

A^∞, Adh(L), is defined by $\{x \in A^\omega \mid Pref^{fin}(x) \subseteq Pref^{fin}(L)\}$. A language $L \subseteq A^\infty$ is *closed*[1] iff $Adh(L) \subseteq L$. A language $L \subseteq A^\infty$ is *prefix closed* iff $Pref(L) = L$.

L^{fin} denotes the *finite* part of L, that is $L^{fin} = L \cap A^*$. L^{inf} denotes the *infinite* part of L, that is $L^{inf} = L \cap A^\omega$. A language is an *ideal* if $L^{fin} = Pref^{fin}(L)$. Every prefix closed language in A^∞ is an ideal and vice-versa[2].

Example. Let us consider $A = \{a, b\}$, $L_1 = \{a^n \mid n \geq 0\}$, $L_2 = \{a^n \mid n \geq 0\} \cup \{a^\omega\}$, $L_3 = \{(ab)^n \mid n \geq 0\}$. Then $Adh(L_1) = \{a^\omega\}$, $Adh(L_2) = \{a^\omega\}$, $Adh(L_3) = \{(ab)^\omega\}$. L_1 and L_3 are not closed, while L_2 is. L_1 and L_2 are ideals and prefix closed.

Transition Systems

Transition systems [Kel76], both labelled and unlabelled, have been widely adopted as means of providing semantics for computations. We shall consider labelled transition systems only, which we shall simply call transition systems.

Definition. (Transition system)

A *transition system* is a a triple $(Q, A, \rightarrow)$ where

(i) Q is a (countable) set of *states* (or *configurations*),

(ii) A is a (finite[3]) set of *action labels*,

(iii) $\rightarrow \subseteq Q \times A \times Q$ is a *transition relation*; $q \rightarrow^a q'$ means that transition a is possible in the state q and the resulting state is q'.

Both Q and A are assumed to be non-empty. Clearly, every discrete system can be represented, at some level of abstraction, as a transition system. Transitions represent

---

1 This definition of closure satisfies the axioms of topological closure. Alternatively, adherence could be defined by means of the ultrametric [BoN79] $d(x,y) = 2^{-n}$, where n is the longest common prefix of x and y.

2 We shall require this distinction when generalizing string languages to obtain trace languages.

3 Finite presentation of programs is assumed.

atomic [4], *indivisible*, *instantaneous* actions. It is assumed that an action represents a *single event*, and if there is any possibility of simultaneous events occurring, such an occurrence can be represented as a sequence of occurrences of events in some arbitrary order. For $a \in A$, $q \in Q$, $q \rightarrow^a$ is used to denote a is *applicable* (i.e. enabled) in the state q. When, for some actions a and b, $q \rightarrow^a$ and $q \rightarrow^b$, we say that actions a and b are *simultaneously enabled*, that is potentially concurrent or non-deterministic.

Note:

- The following need *not* hold:

  $$\forall q \in Q \; \exists a \in A: q \rightarrow^a.$$

  A state q is a *terminal* state iff for all $a \in A$ such that $q \not\rightarrow^a$.

- Two differently labelled transitions applied in the same state may have the same direct successor, i.e. the following is allowed:

  $$q \rightarrow^a q' \wedge q \rightarrow^b q' \wedge a \neq b.$$

- We assume $\rightarrow^a$ is a (partial) function $Q \times A \rightarrow Q$, i.e. no two identically labelled transitions, when applied in the same state, may result in two different successors (this is called *unambiguity*):

  $$(q \rightarrow^a q' \wedge q \rightarrow^a q'') \Rightarrow q' = q''.$$

- A transition system is called *finitely branching* iff

  $$B_q = \{(a,q') \in A \times Q \mid q \rightarrow^a q'\} \text{ is finite}$$

  for all $q \in Q$. We shall only consider finitely branching systems, which is guaranteed by the assumptions of unambiguity and the finiteness of A.

Topology

This section is based on [Kur66] [Smy83].

---

4 All models known to the author, apart from [Lam86], assume atomicity.

A *topology* on a set S is a collection of subsets of S that contains $\varnothing$ and S, and is closed under finite intersection and arbitrary union. A set together with a topology $\mathfrak{T}$ on S is called a *topological space*; the elements of $\mathfrak{T}$ are the *open sets* of the space. A *base* of the topology is a subset $\mathfrak{B} \subseteq \mathfrak{T}$ such that every open set is the union of elements of $\mathfrak{B}$. A set X is *closed* iff its complement S-X is open. A set is *dense* iff its closure Cl(X) is S. A G_δ*-set* is a countable (finite or infinite) intersection of a family of open sets.

The closure operation Cl for any $X \subseteq S$ satisfies the following axioms:

1) $Cl(X \cup Y) = Cl(X) \cup Cl(Y)$
2) $X \subseteq Cl(X)$
3) $\varnothing = Cl(\varnothing)$
4) $Cl(Cl(X)) = Cl(X)$.

In addition, if:

5) $Cl(\{p\}) = \{p\}$, for $p \in S$

then the space is $\mathfrak{T}_1$.

A space is *Hausdorff* (or a $\mathfrak{T}_2$-space) iff for each pair of points $p \neq q$ there exist two open sets G, H such that $p \in G$, $q \in H$ and $G \cap H = \varnothing$.

Let (P, ≤) be a poset. The *Alexandroff topology* over P is the collection of suffix closed sets, that is sets $X \subseteq P$ such that $X = \uparrow X$. (A topological space with the Alexandroff topology is non-Hausdorff).

Let (P, ≤) be a cpo. The *Scott topology* is defined as follows. A set X is Scott-open iff it is suffix-closed and, for every directed set $Y \subseteq P$, if lub(Y) ∈ X then some element of Y is in X. (A topological space with the Scott topology is non-Hausdorff).

Example. Let us consider the domain A^∞ with prefix ordering ≤. For A = {a, b}, examples of Alexandroff-open sets are $X_1 = \{a^n \mid n \geq 0\} \cup \{a^\omega\}$, $X_2 = \{ab^n \mid n \geq 0\} \cup \{b^\omega, ab^\omega\}$, and $X_3 = \{a^\omega\}$. X_1 is also Scott-open, but X_2 and X_3 are not. The base of the Alexandroff topology is $\{\uparrow x \mid x \in A^\infty\}$. The base of the Scott topology is $\{\uparrow x \mid x \in A^*\}$. In order to show that Alexandroff topology is non-Hausdorff, let us take p = {a} and q = {aa}. The same pair of points serves as a counter-example for Scott topology being Hausdorff.

3.2. Trace Languages

Trace languages [Maz77] are an attempt to introduce additional structure into formal languages in order to provide a mathematical description of non-sequential behaviours. It is assumed in trace theory that observations are sequential in nature, but, for systems that exhibit concurrency, different external observers may disagree on the ordering of *concurrent* events. These differences are subjective, as they depend on the position of an observer, or the actual timing of the execution, and thus should be considered irrelevant. On the other hand, the ordering of *causally related* events is objective, that is independent of the observer, and should, therefore, be distinguished.

Trace languages are derived from sequential languages. The additional structure is given in terms of an independency relation, which describes causal relationship of actions within a system. Finitary trace languages are due to Mazurkiewicz [Maz77]. (Finitary) trace languages were considered in [Bed87]. In [Maz84b], a modular decomposition of trace languages, forming a basis of a process algebra, was formally derived. The questions of the relationship of the various classes of trace languages were addressed in [AaR88].

Dependence graphs [AaR88] are a related notion.

Independency and Trace Equivalence

Let A denote a (finite) alphabet. By a *concurrent alphabet* we shall mean an ordered pair:

$$C = (A, \iota)$$

where A is called the *alphabet* of C and $\iota \subseteq A \times A$ is a symmetric and irreflexive relation (the *independency*).

Intuitively, the independency relation describes possible concurrency within a system; irreflexivity prohibits an action to be concurrent with itself (i.e. $a \iota b \Rightarrow a \neq b$), and symmetry requires that concurrency is always mutual. Note that independency is not, in general, transitive.

Obviously, $\varnothing$ and $A \times A$ are independencies (called *empty* and *full* independency correspondingly). An empty independency corresponds to a sequential (but possibly non-deterministic) system.

From now on we assume A is fixed. We define *trace equivalence* [Maz77], denoted $\equiv_\iota^*$, as the least congruence in the monoid of strings $S = (A^*, ., \varepsilon)$ such that:

$$a \iota b \Rightarrow ab \equiv_\iota^* ba.$$

It follows then for all $w', w'' \in A^*$:

$$w' \equiv_\iota^* w'' \quad \Leftrightarrow \quad \begin{array}{l} \exists\ w_0, w_1, \ldots w_n \ \ n \geq 0:\ w_0 = w',\ w_n = w'' \text{ and} \\ \forall\ k, 1 \leq k \leq n, \exists\ u, v \in A^*, a, b \in A: \\ a \iota b \text{ and } w_{k-1} = uabv,\ w_k = ubav. \end{array}$$

By definition, trace equivalence is a congruence with respect to concatenation. This congruence seems quite strong: note that $w' \equiv_\iota^* w'' \Rightarrow |w'| = |w''|$.

Intuitively, trace equivalence allows to permute any two consecutive actions, providing they are independent. This is why trace equivalence is sometimes referred to as the *permutation equivalence* [BoC88]. Note that when independency is empty, no two actions may be permuted, which corresponds to our understanding of sequential systems.

The main objective of trace equivalence is to equate all, and those strings only, which differ in the order of concurrent actions, that is, their ordering can be viewed as irrelevant. The independency relation, as opposed to the compatibility relation in event structures [Win86], is a syntactical notion, and it can usually be determined from the syntactical structure of processes in a concurrent language.

Example. Let us consider $A = \{a, b, c\}$, with $\iota = \{(a,b), (b,a)\}$. Then $abc \equiv_\iota^* bac$, but it is *not* the case that $abc \equiv_\iota^* acb$. Using a process algebra notation, strings abc and bac intuitively correspond to the behaviour of a process (aNIL || bNIL);cNIL (note that acb is not admissible here). Also, $aab \equiv_\iota^* aba \equiv_\iota^* baa$, which, similarly, is a representation of the behaviour of (aaNIL || bNIL). On the other hand, if $\iota = \varnothing$, then all actions are causally related and cannot be permuted. This corresponds to processes like abcNIL + bacNIL, in which case strings abc and bac would *not* be equivalent.

Traces

Equivalence classes over C are called *traces*. A trace generated by a string w will be denoted $[w]_\iota$ or [w] if ι understood.

For each (finitary) language $L \subseteq A^*$ define:

$$[\, L \,]_\iota = \{[w]_\iota \mid w \in L\}.$$

The set

$$\Theta_\iota{}^* = [\, A^* \,]_\iota$$

is the set of all (finite) traces over the concurrent alphabet $C = (A, \iota)$. The quotient algebra $T^* = (A^*, ., \varepsilon)/_{\equiv}$ is called the *algebra of traces* over (A, ι). T^* is a monoid [Maz84a] (sometimes called a free partially commutative monoid). When $\iota = \varnothing$ the algebra of traces is isomorphic to the algebra of strings over A.

The following equalities [Maz88] hold for $u, w \in A^*, \tau, \tau_1, \tau_2, \tau_3, \sigma', \sigma'' \in \Theta_\iota{}^*$:

$[u]\,[w] = [uw]$
$\tau\,[\varepsilon] = [\varepsilon]\,\tau = \tau$
$\tau_1(\tau_2\,\tau_3) = (\tau_1\tau_2\,)\tau_3$
$\sigma'\,\tau_1\sigma'' = \sigma'\,\tau_2\sigma'' \;\Rightarrow\; \tau_1 = \tau_2\,.$

The first of the above laws states that, in order to concatenate two traces, it is sufficient to concatenate any pair of the representants and then take the corresponding equivalence class.

Example. Let us consider A = {a, b, c}, with ι = {(a,b), (b,a)}. Then [abc] = {bac, abc}, [aab] = {aab, aba, baa}. Intuitively, trace equivalence groups together all possible sequentializations of the given behaviour, for example the trace [abc] represents (aNIL || bNIL);cNIL. On the other hand, abcNIL + bacNIL would give rise to two distinct traces, [abc] and [bac].

Example. For L = {ab, abab, baba} with a ι b we have [L] = {[ab, ba], [abab, abba, aabb, bbaa, baba, baab]}.

Trace Prefix Ordering

A trace σ is a *prefix* of trace τ over $C = (A, \iota)$ [Maz88] if there is γ such that $\tau = \sigma\gamma$ (that is, σ can be extended to τ). Trace prefix ordering will be denoted $\leq_\iota$ (or $\leq$ if ι understood). The set of all prefixes of a trace τ will be denoted by $\mathrm{Pref}(\tau)$. The set Θ_ι^* of all traces over $C = (A, \iota)$ is ordered by $\leq_\iota$ defined as:

$$\tau_1 \leq \tau_2 \;\Leftrightarrow\; \tau_1 \in \mathrm{Pref}(\tau_2).$$

$(\Theta_\iota^*, \leq)$ is a poset. The set of prefixes of a given trace is directed, but not a total order like in the algebra of strings. Relation $\leq$ will be referred to as the *dominating* relation in Θ_ι^*. If $\tau_1 \leq \tau_2$ we say that τ_1 is *dominated* by τ_2. If there is a trace that dominates both τ_1 and τ_2 we say that τ_1 and τ_2 are *consistent*; otherwise they are *inconsistent*. If $\tau_1 \leq \tau_2$ or $\tau_2 \leq \tau_1$ we say that τ_1 and τ_2 are *comparable*; otherwise they are *incomparable*. Unlike in the algebra of strings, two incomparable traces may be consistent.

Example. Let us consider traces over $C = (A, \iota)$, where $A = \{a, b, c, d\}$ and $\iota = \{(a,b), (b,a), (a,c), (c,a)\}$. Then $[\varepsilon]$ is dominated by [a]; [a] is dominated by [ab] and [ad]; [b] is dominated by [ab], but not [ad]. Also, [a] and [b] are consistent but incomparable, whereas [ab] and [ad] are inconsistent.
Let w = abcad. Fig. 3.2.1 represents Pref([w]), that is the poset of all prefixes of [w].

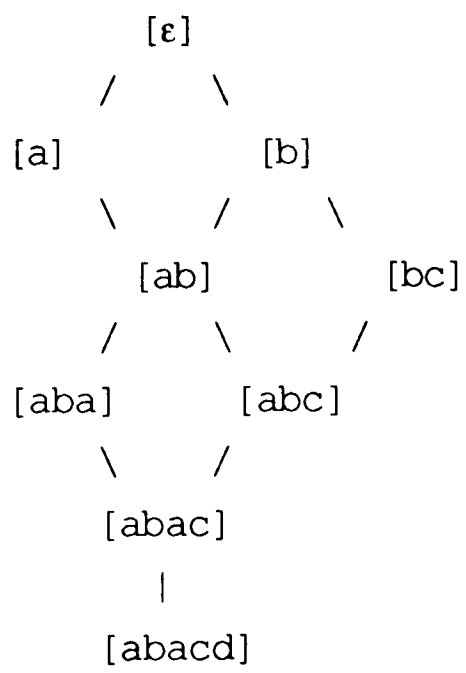


Fig. 3.2.1. The poset of all prefixes of [abcad] with a ι b, a ι c.

Trace prefix ordering can be viewed as ordering on partial executions, where each trace corresponds to a (global) state. $[\varepsilon]$ is the initial state, incomparable prefixes of the same trace represent states arising during concurrent execution, and $\tau_1 \leq \tau_2$ means that τ_1 is a partial execution leading to τ_2. Inconsistent states are the effect of non-determinism (conflict resolution).

Left Cancellation for Finite Traces

The following binary left cancellation operator, denoted /: $\Theta_\iota^* \times \Theta_\iota^* \rightarrow \Theta_\iota^*$, may now be defined:

$$\forall\ \sigma, \gamma \in \Theta_\iota^*: \qquad \sigma/\gamma = \tau \ \Leftrightarrow\ (\gamma \leq_\iota \sigma \wedge \gamma\tau = \sigma).$$

σ/γ is pronounced σ *after* γ; it denotes the continuation of σ after its prefix γ has been completed. Observe that σ/γ is well defined and equal $[x/y]_\iota$ for some $x \in \sigma$ and $y \in \gamma$ such that $y \leq x$ (the existence of such x, y is a consequence of Observation 3.2.9).

We shall be concerned with extending left cancellation onto Θ_ι^∞ in later sections.

Decomposition of Finite Traces

We can extend the independency relation onto finite traces. For a trace $\sigma \in \Theta_\iota^*$ define the set of actions of σ, denoted $Act(\sigma)$, as follows:

$$Act(\sigma) = \{a \in A \mid xay \in \sigma \text{ for some } x, y \in A^*\}.$$

Two traces $\tau_1, \tau_2 \in \Theta_\iota^*$ are *independent* if, and only if, $Act(\tau_1) \times Act(\tau_2) \subseteq \iota$ and *dependent* otherwise. A trace $\tau \in \Theta_\iota^*$ is called *connected* if any two non-empty traces τ_1, $\tau_2 \in \Theta_\iota^*$ with $\tau = \tau_1\tau_2$ are dependent.

Definition [Maz88]. A *decomposition* of a trace $\tau \in \Theta_\iota^*$, denoted $\Delta(\tau)$, is a set $\{\tau_1, \tau_2, ..., \tau_n\}$ such that:

(i) $\tau = \tau_1\tau_2...\tau_n$

(ii) τ_i is non-empty and connected for each i, $1 \leq i \leq n$

(iii) τ_i, τ_j are independent for each i, j distinct, $1 \leq i \leq n$, $1 \leq j \leq n$.

Elements of $\Delta(\tau)$ are called *independent components* of trace τ.

Example. Let $b \, \iota \, c$. Then:

$\Delta([abb]) = \{[abb]\}$
$\Delta([cbbcc]) = \{[bb], [ccc]\}$
$\Delta([abc]) = \{[abc]\}$.

Clearly, if $\iota = \varnothing$ then $\Delta(\tau) = \{\tau\}$.

Finitary Trace Languages

Let $C = (A, \iota)$ be a concurrent alphabet. Each subset of $\Theta_\iota{}^*$ of all traces over C is called a (finitary) *trace language* [Maz77] [Maz88]. Concatenation of finitary trace languages is defined as follows. Let T_1, T_2 be finitary trace languages. Then:

$$T_1T_2 = \{\tau_1\tau_2 \mid \tau_1 \in T_1, \tau_2 \in T_2\}.$$

Trace iteration is defined in the usual way, that is:

$$T^* = \bigcup \{T^n \mid n \in N\}$$

where:

$$T^0 = \{[\varepsilon]\}$$
$$T^{n+1} = T^nT.$$

For each trace language T define:

$$Pref(T) = \{Pref(\tau) \mid \tau \in T\}.$$

A trace language T is *prefix closed* if T = Pref(T). A trace language T is *directed* if $(T, \leq)$ is a directed set, that is it is non-empty and:

$$\tau_1 \in T, \ \tau_2 \in T \Rightarrow \exists \, \tau \in T: \tau_1 \leq \tau \wedge \tau_2 \leq \tau.$$

Properties of finitary trace languages have been investigated in a number of papers. It turns out that most properties known for string languages generalize onto trace languages. For example, although the definition of trace iteration included does not allow for the generalization of Kleene theorem for strings, it is possible to define iteration in such a way

that regular trace languages can be characterized as the least class of languages closed under union, concatenation and iteration [Maz88] (generalization of Kleene theorem for strings). A trace language T is *regular* iff the family of sets:

$$\{T/\tau \mid \tau \in \Theta_\iota^*\}$$

is finite, where T/τ denotes the set $\{\sigma \in \Theta_\iota^* \mid \tau\sigma \in T\}$.

The iteration of a trace language T is defined by $(\Delta(T))^*$ [Maz88]. The relationship of regular, context-free and context-sensitive trace languages and graph grammars is established in [AaR88].

Decomposition of Finitary Trace Languages

Decomposition can also be extended onto finitary trace languages. Let $T \subseteq \Theta_\iota^*$. Then the *decomposition* of T, denoted by $\Delta(T)$, is defined:

$$\Delta(T) = \bigcup \{\Delta(\tau) \mid \tau \in T\}.$$

Proposition 3.2.1. [Maz88]. The following are properties of decomposition of trace languages.

(i) $T^* \subseteq \Delta(T)^*$,

(ii) $T = \Delta(T)$ if $\iota = \varnothing$,

(iii) $\Delta(\Delta(T)) = \Delta(T)$.

The language T is *connected* if, and only if, $T = \Delta(T)$.

Infinite Traces

We can extend trace equivalence $\equiv_\iota^*$ in the monoid of finite strings $S = (A^*, ., \varepsilon)$ onto A^∞ using a definition similar to the permutation equivalence introduced in [BoC88]. We achieve this by weakening string prefix ordering $\leq$, denoted for the purpose of this section by $\leq_S$, to a *trace preorder* relation on A^∞, which abstracts from irrelevant interleavings of consecutive independent symbols. Finite or infinite number of permutations is allowed.

The required relation is arrived at in two steps. First, we introduce trace preorder $\leq^*$ on A^*, which is then shown to determine $\equiv_\iota{}^*$. Later, we extend trace preorder onto A^∞ (denoted $\leq^\infty$). Finally, infinite traces are defined as equivalence classes with respect to the equivalence $\equiv_\iota{}^\infty$ determined by the preorder $\leq^\infty$.

Definition. (Trace preorder on A^*). Let $x, y \in A^*$, then :

$$x \leq^* y \;\Leftrightarrow\; (\exists\, z \in A^* :\; x \leq_S z \;\wedge\; z \equiv_\iota{}^* y\,).$$

Observation 3.2.2. $x \leq_S y \Rightarrow x \leq^* y$.

Proposition 3.2.3. $\leq^*$ is a preorder.

Proof.

1) Reflexivity. $x \leq^* x$ follows trivially from reflexivity of $\leq_S$ and $x = y \Rightarrow x \equiv_\iota{}^* y$.

2) Transitivity. Suppose $x \leq^* y \;\wedge\; y \leq^* z$. Then from definition of $\leq^*$ we have:

(2a) $\exists$ u: $x \leq_S u \;\wedge\; u \equiv_\iota{}^* y$, and

(2b) $\exists$ w: $y \leq_S w \;\wedge\; w \equiv_\iota{}^* z$.

We need to show the existence of v such that $x \leq_S v \;\wedge\; v \equiv_\iota{}^* z$. We construct v as follows:

$$v = xu'w'$$

where u' denotes u/x (must exist because $x \leq_S u$), and w' denotes w/y (again, this exists because $y \leq_S w$). We now have:

$$[v] = [xu'w'] =^\blacklozenge [xu'][w'] = [u][w'] = [y][w'] =^\blacklozenge [yw'] = [w].$$

(where steps marked with $\blacklozenge$ follow from $[x][y] = [xy]$). We have thus shown that $v \equiv_\iota{}^* w \equiv_\iota{}^* z$, and hence we have constructed v such that $x \leq_S v \wedge v \equiv_\iota{}^* z$. We can conclude that $x \leq^* z$.

❑

Example. For $A = \{a, b\}$ and $a \,\iota\, b$, we have that $\varepsilon \leq^* a$, $a \leq^* ab$, $a \leq^* ba$, $ab \leq^* ba$, and $ba \leq^* ab$. However, when $\iota = \varnothing$, it is *not* the case that $ab \leq^* ba$.

Observation 3.2.4. Note that for finite strings over A we have $x \equiv_\iota{}^* y$ iff $x \leq^* y \wedge y \leq^* x$ (proof follows directly from definition of $\leq^*$).

We now extend the above preorder to a preorder $\leq^\infty$ on A^∞. We require that for every finite prefix x of u there must exist a finite prefix y of v such that $x \leq^* y$.

Definition. (Trace preorder on A^∞). Let $u, v \in A^\infty$, then :

$$u \leq^\infty v \Leftrightarrow (\forall x \in Pref^{fin}(u)\ \exists y \in Pref^{fin}(v):\ x \leq^* y).$$

Proposition 3.2.5. $\leq^\infty$ is a preorder.

Proof.

1) Reflexivity. $u \leq^\infty u$ obvious (follows from reflexivity of $\leq_S$ and $\leq^*$).

2) Transitivity. Suppose $u \leq^\infty v \wedge v \leq^\infty w$. Then from $u \leq^\infty v$ and the definition of $\leq^\infty$ we have: $\forall x \in Pref^{fin}(u)\ \exists y \in Pref^{fin}(v):\ x \leq^* y$. Since $v \leq^\infty w$, we deduce from the definition of $\leq^\infty$ that $\exists z \in Pref^{fin}(w):\ y \leq^* z$. The final conclusion, $x \leq^* z$, follows from transitivity of $\leq^*$.

❑

Example. Let us again consider $A = \{a, b\}$ with $a\ \iota\ b$. It is easy to notice that $a \leq^\infty a^\omega$, $a^\omega \leq^\infty (ab)^\omega$, and $a^\omega \leq^\infty (ba)^\omega$.

The preorder $\leq^\infty$ determines an equivalence relation $\equiv_\iota{}^\infty$ which coincides with trace equivalence $\equiv_\iota{}^*$ on A^*.

Proposition 3.2.6. $\leq^\infty$ and $\leq^*$ coincide on A^*.

Proof.

1) Suppose $x \leq^* y$ for some $x, y \in A^*$. We need to show that $\forall m \in Pref^{fin}(x)\ \exists n \in Pref^{fin}(y)$: $m \leq^* n$. Suppose $m \leq_S x$, then, obviously, $m \leq^* x$, and from the transitivity of $\leq^*$, we deduce $m \leq^* y$. We have thus shown, for any prefix m of x, the existence of $n = y$ such that $m \leq^* n$, which concludes the proof.

2) Suppose $x \leq^\infty y$ for some $x, y \in A^*$. We need to show $x \leq^* y$. Take $m = x$, then, from definition of $\leq^\infty$, $\exists n \in Pref^{fin}(y)$: $x \leq^* n$. Since $n \leq_S y \Rightarrow n \leq^* y$, it follows from transitivity of $\leq^*$ that $x \leq^* y$.

❑

Unlike the definition of trace equivalence for finite strings, the definition of $\equiv_\iota{}^\infty$ allows for an *infinite* number of permutations of two consecutive independent symbols. For example, $(ab)^\omega \equiv_\iota{}^\infty (ba)^\omega$ if $a\ \iota\ b$. The infinite trace $[(ab)^\omega]$ is thus a representation of the behaviour of the process (fix X.aX) || (fix X.bX). This process must execute both a and b infinitely often. It is easy to see that $[(ab)^\omega]$ subsumes all, and only those sequences over {a, b} that contain an infinite number of both a's and b's (a fairmerge of a^ω and b^ω [Par81]). On the

other hand, the behaviour of the process fix X.(aX + bX), where a and b are dependent, would be represented by a set of equivalence classes that are singleton sets $[(a^*b^*)^\omega]$. Note that, in this case, a finite number of occurrences of either a or b is allowed.

We have thus extended trace equivalence onto A^∞. We shall denote trace equivalence in A^∞ by $\equiv_\iota$ (the subscript will be omitted if ι understood). A trace generated by a string w will be denoted $[w]_\iota$ or [w].

For each infinitary language $L \subseteq A^\infty$ define:

$$[\, L \,]_\iota \;=\; \{[w]_\iota \mid w \in L\}.$$

The set

$$\Theta_\iota{}^\infty = [\, A^\infty \,]_\iota$$

is the set of all (finite or infinite) traces over the concurrent alphabet $C = (A, \iota)$. The set of infinite traces, denoted $\Theta_\iota{}^\omega$, is defined as $[\, A^\omega \,]_\iota$.

Trace Prefixes for Infinite Traces

Trace prefix ordering on infinite traces, denoted $\leq_\iota$, is determined by the preorder $\leq^\infty$. A (finite or infinite) trace σ is a *prefix* of a (finite or infinite) trace τ over $C = (A, \iota)$ iff:

$$\forall\, x \in \sigma\ \exists\, y \in \tau\colon\ x \leq^\infty y.$$

Let us denote the above relation over $\Theta_\iota{}^\infty$ by $\leq_\iota{}^\infty$, and trace prefix ordering over $\Theta_\iota{}^*$ by $\leq_\iota{}^*$. We now prove that $\leq_\iota{}^\infty$ coincides with $\leq_\iota{}^*$ over $\Theta_\iota{}^*$. We shall need the following lemma.

Lemma 3.2.7. $\forall\ x, y, u, w \in A^*$:

$$(x \leq_S y,\ u \leq_S w,\ x \equiv_\iota{}^* u,\ y \equiv_\iota{}^* w) \;\Rightarrow\; (y/x \equiv_\iota{}^* w/u).$$

Proof.
$y \equiv_\iota{}^* w \;\Rightarrow\; x(y/x) \equiv_\iota{}^* u(w/u)$ (from definition of /) $\;\Rightarrow\; [x(y/x)] = [u(w/u)] \;\Rightarrow$ $[x][y/x] = [u][w/u]$ (by definition of concatenation) $\;\Rightarrow\; [x][y/x] = [x][w/u]$ (because $x \equiv_\iota{}^* u$) $\;\Rightarrow\; [y/x] = [w/u]$ (by left cancellation) $\;\Rightarrow\; y/x \equiv_\iota{}^* w/u.$

❑

Proposition 3.2.8. $\leq_\iota^\infty$ coincides with $\leq_\iota^*$ over Θ_ι^*.

Proof.

1) Suppose $\sigma, \tau \in \Theta_\iota^*$ such that $\sigma \leq_\iota^* \tau$. Then, from the definition of $\leq_\iota^*$, there exists γ such that $\sigma\gamma = \tau$. Since $\leq^\infty$ and $\leq^*$ coincide on A^*, it is sufficient to show that for every representant x of σ there exists a representant y of τ such that $x \leq^* y$. We have $[x]\,[y] = [z] = [xy]$, for any representants $x \in \sigma$, $y \in \gamma$ and $z \in \tau$. Since $x \leq_S xy$ $\Rightarrow x \leq^* xy$ and $xy \equiv_\iota^* z$, we conclude $x \leq^* z$. We have thus shown $x \leq^\infty z$.

2) Suppose $\sigma, \tau \in \Theta_\iota^*$ such that $\sigma \leq_\iota^\infty \tau$. We need to show the existence of $\gamma \in \Theta_\iota^*$ such that $\sigma\gamma = \tau$. By definition of $\leq_\iota^\infty$ we have $\forall\, x \in \sigma\ \exists\, y \in \tau\colon\ x \leq^\infty y$, and, since $\leq^\infty$ agrees with $\leq^*$ on A^* by Proposition 3.2.6, it follows that $\forall\, x \in \sigma\ \exists\, y \in \tau\colon\ x \leq^* y$. From definition of $\leq^*$ we have $\forall\, x \in \sigma\ \exists\, y \in \tau\colon\ (\exists\, z\colon\ x \leq_S z \wedge z \equiv_\iota^* y)$. Thus, $\forall\, x \in \sigma\ \exists\, z \in \tau\colon\ x \leq_S z$ and $|\,x\,| \leq |\,z\,|$. Then for any $x_1, x_2 \in \sigma\ \exists\, z_1, z_2 \in \tau\colon x_1 \leq_S z_1$ and $x_2 \leq_S z_2$. Define $\gamma = [z_1/x_1]$. By Lemma 3.2.7, $\gamma = [z_2/x_2]$; hence the construction of γ does not depend on the choice of representants. Now, $\sigma\gamma = [x_1][z_1/x_1] = [x_1(z_1/x_1)] = [z_1] = [y] = \tau$, which concludes the proof.

❑

We have thus extended trace prefix ordering onto Θ_ι^∞. From now on we shall denote trace prefix ordering on Θ_ι^∞ by $\leq_\iota$ (or $\leq$ if ι understood).

Infinite traces can be viewed as limits of chains of finite traces (or least upper bounds of directed and prefix closed finitary trace languages). Finite prefixes of infinite traces correspond to finite approximations. Note that, in contrast to prefix ordering for strings, it is possible to show two distinct infinite traces, of which one dominates the other one. For example, the following holds for a ι b:

$$[(a)^\omega] \leq [b(a)^\omega] \leq \ \ldots\ \leq [(ab)^\omega].$$

Observation 3.2.9. Note that a direct consequence of the definition of $\leq^*$ is that, whenever $\sigma \leq_\iota \tau$, for $\sigma, \tau \in \Theta_\iota^*$, then for all $x \in \sigma$ there exists $y \in \tau$ such that $x \leq_S y$. This does *not* extend onto Θ_ι^∞, for example $[a^\omega] \leq_\iota [ba^\omega]$ when a ι b, but for no $y \in [ba^\omega] = a^*ba^\omega$ do we have $a^\omega \leq_S y$.

The set of all prefixes of a trace τ will be denoted by $\mathrm{Pref}(\tau)$; the set of finite and infinite prefixes will be denoted $\mathrm{Pref}^{\mathrm{fin}}(\tau)$ and $\mathrm{Pref}^{\mathrm{inf}}(\tau)$ respectively. Obviously:

$\tau_1 \leq_\iota \tau_2 \;\Leftrightarrow\; \tau_1 \in \mathrm{Pref}(\tau_2)$.

The definitions of traces that are consistent, inconsistent, comparable, incomparable, and a trace dominated by another trace are defined analogously.

Example. Let A = {a, b, c}, ι = {(a,b), (b,a)}, and $\tau = [(ab)^2c^\omega]$. The set of all prefixes of τ is shown in Fig. 3.2.2.

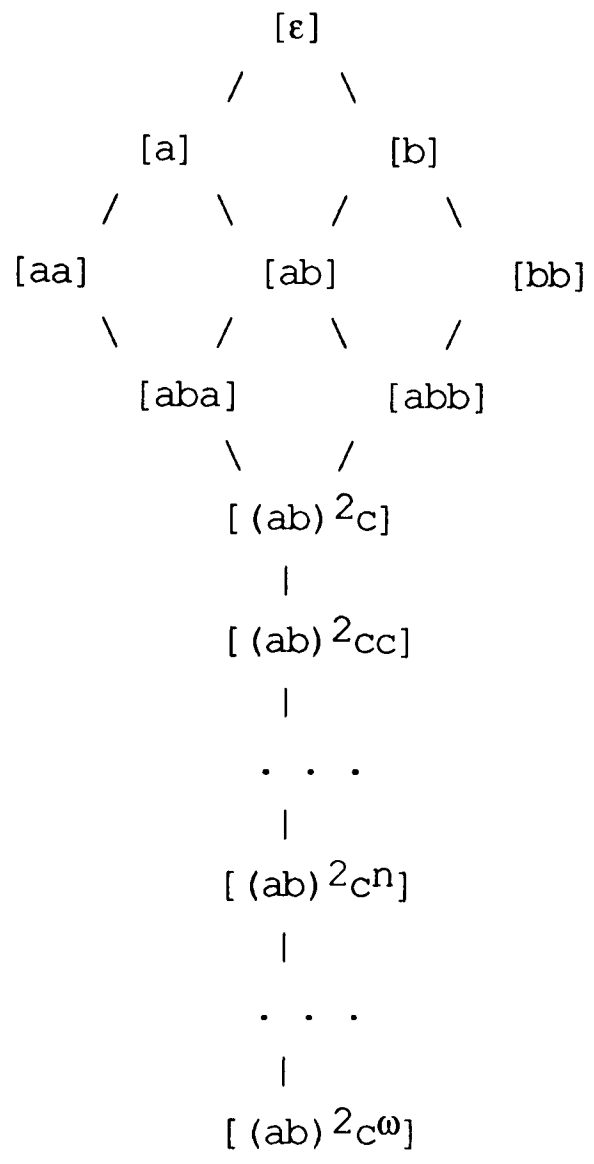


Fig. 3.2.2. The poset of all prefixes of $[(ab)^2c^\omega]$, a ι b.

A trace τ is *maximal* in Θ_ι^∞ if it is maximal with respect to the dominating relation $\leq_\iota$, that is there does not exist a trace $\sigma \neq \tau$ such that $\tau \leq_\iota \sigma$. The set of all traces that are maximal in Θ_ι^∞ is denoted $\mathrm{Max}(\Theta_\iota^\infty)$. Maximal infinite traces correspond to the *global*, objective view of the system behaviour, while the non-maximal ones represent the *local*, therefore subjective, view of an agent. No finite trace is maximal in Θ_ι^∞.

Decomposition of Infinite Traces

We now define decomposition of infinite traces. Infinite traces may be viewed as least upper bounds of directed sets of finite traces. In order to decompose an infinite trace, we need to decompose the set of its finite prefixes, and take least upper bounds of all maximal connected directed sets M contained in this decomposition. Formally, for $\sigma \in \Theta_\iota{}^\omega$, we define $\Delta(\sigma)$ as follows:

$$\Delta(\sigma) = \{\mathrm{lub}(M) \mid M \text{ maximal directed subset of } \Delta(\mathrm{Pref}^{\mathrm{fin}}(\sigma)) \text{ s.t. } \Delta(M) = M\}.$$

This definition relies on the existence of least upper bounds of directed trace languages (Theorem 3.5.3). Note that $\Delta(\sigma)$ is non-empty. Also, if $\iota = \varnothing$ then $\Delta(\tau) = \{\tau\}$.

Example. Let $b\ \iota\ c$. Then:

$$\Delta([ab^\omega]) = \{\mathrm{lub}\{[\varepsilon],[ab^*]\}\} = \{[ab^\omega]\}$$
$$\Delta([cb^\omega]) = \{\mathrm{lub}\{[\varepsilon],[c]\}, \mathrm{lub}\{[b^*]\}\} = \{[c], [b^\omega]\}$$
$$\Delta([abc^\omega]) = \{\mathrm{lub}\{[\varepsilon],[a],[ab],[ac],[abcc^*]\}\} = \{[abc^\omega]\}.$$

Infinitary Trace Languages

Let $C = (A, \iota)$ be a concurrent alphabet. Each subset of $\Theta_\iota{}^\infty$ of all traces over C is called an *infinitary trace language.*

For each trace language T define:

$$\mathrm{Pref}(T) = \{\mathrm{Pref}(\tau) \mid \tau \in T\}.$$

$\mathrm{Pref}^{\mathrm{fin}}(T)$ and $\mathrm{Pref}^{\mathrm{inf}}(T)$ are defined as $\mathrm{Pref}(T) \cap \Theta_\iota{}^*$ and $\mathrm{Pref}(T) \cap \Theta_\iota{}^\omega$ respectively. A trace language T is *prefix closed* if $T = \mathrm{Pref}(T)$. A trace language T is *directed* if $(T, \leq_\iota)$ is a directed set. An *adherence* of a trace language $T \subseteq \Theta_\iota{}^\infty$, Adh(T), is defined by $\{\tau \in \Theta_\iota{}^\omega \mid \mathrm{Pref}^{\mathrm{fin}}(\tau) \subseteq \mathrm{Pref}^{\mathrm{fin}}(T)\}$. A trace language $T \subseteq \Theta_\iota{}^\infty$ is *closed* [5] iff $\mathrm{Adh}(T) \subseteq T$.

Note that $\Theta_\iota{}^\infty$ is a closed language, but $\Theta_\iota{}^* \cup \mathrm{Max}(\Theta_\iota{}^\infty)$ is not, in general, closed. Also, $\mathrm{Adh}(\Theta_\iota{}^\infty) = \Theta_\iota{}^\omega$, but it is *not* the case that $\mathrm{Max}(\Theta_\iota{}^\infty) = \Theta_\iota{}^\omega$. This is in contrast to the

---

5 This notion of closure satisfies axioms 1-4 of topological closure.

string algebra A^{∞}, where A^{ω} is exactly the set of all maximal sequences with respect to string prefix ordering and adherence of A^{∞}.

T^{fin} denotes the *finite* part of T, that is $T^{fin} = T \cap \Theta_{\iota}^*$. T^{inf} denotes the *infinite* part of T, that is $T^{inf} = T \cap \Theta_{\iota}^{\omega}$. A trace language is an *ideal* if $T^{fin} = Pref^{fin}(T)$.

Note that Θ_{ι}^{∞} is an ideal, and so is $\Theta_{\iota}^* \cup Max(\Theta_{\iota}^{\infty})$. This is because $Pref^{fin}(Max(\Theta_{\iota}^{\infty})) = \Theta_{\iota}^*$, which is a direct result of the proposition below.

Proposition 3.2.10. $\forall\, \sigma, \tau \in \Theta_{\iota}^{\infty}: \sigma \leq_{\iota} \tau \Leftrightarrow Pref^{fin}(\sigma) \subseteq Pref^{fin}(\tau)$.

Proof.

$\Rightarrow$) Follows from transitivity of $\leq_{\iota}$.

$\Leftarrow$)[6] Follows from algebraicity of Θ_{ι}^{∞} (Theorem 3.5.3).

❑

Proposition 3.2.11. $Pref^{fin}(Max(\Theta_{\iota}^{\omega})) = \Theta_{\iota}^*$.

Proof. Obviously, $Pref^{fin}(\Theta_{\iota}^{\omega}) = \Theta_{\iota}^*$, $Max(\Theta_{\iota}^{\omega}) \subseteq \Theta_{\iota}^{\omega}$. Take any $\sigma \notin Max(\Theta_{\iota}^{\omega})$, then there exists $\tau \in Max(\Theta_{\iota}^{\omega})$ such that $\sigma \leq \tau$. From Proposition 3.2.10, $Pref^{fin}(\sigma) \subseteq Pref^{fin}(\tau)$, hence $Pref^{fin}(\Theta_{\iota}^{\omega} - \{\sigma\}) = Pref^{fin}(\Theta_{\iota}^{\omega})$.

❑

Every prefix closed trace language is an ideal, but the converse does not hold. Every trace language T closed with respect to finite prefixes, that is $T = Pref^{fin}(T)$, is an ideal.

A trace τ is *maximal* in T if there does not exist a trace $\sigma \in T$ such that $\tau \leq_{\iota} \sigma$ and $\tau \neq \sigma$. The set of all traces that are maximal in T is denoted Max(T).

Example. Let us consider trace languages over {a,b} for a ι b. Let:

$T_1 = \{[a^*]\}$,
$T_2 = \{[a^*b^*]\} \cup \{[(a^*b^*)^{\omega}]\}$,
$T_3 = \{[a^*b^*]\} \cup \{[(ab)^{\omega}]\}$, and
$T_4 = \{[(ab)^n]\}$.

---

6 This observation is due to M.W. Shields.

Then Adh(T_1) = {[a^ω]}, Adh(T_2) = {[$(a^*b^*)^\omega$]} = Adh(T_3) = Adh(T_4). T_1, T_3, and T_4 are not closed, but T_2 is closed. T_1, T_2, T_3 and T_4 are directed. T_1, T_2 and T_3 are ideals. T_1 and T_2 are prefix closed, but T_3 and T_4 are not (there are infinite prefixes of [$(ab)^\omega$], for example [a^ω], which are not contained in T_3). [$(ab)^\omega$] is maximal in T_2 and T_3.

Fig. 3.2.3 shows the set of all prefixes of the language T = {[$(abc)^*$abd]}. We assume A = {a, b, c, d}, ι = {(a,b), (b,a)}. An example of a process that gives rise to this trace language is: p = fix X.((aNIL || bNIL);p'), where p' = cX + dNIL.

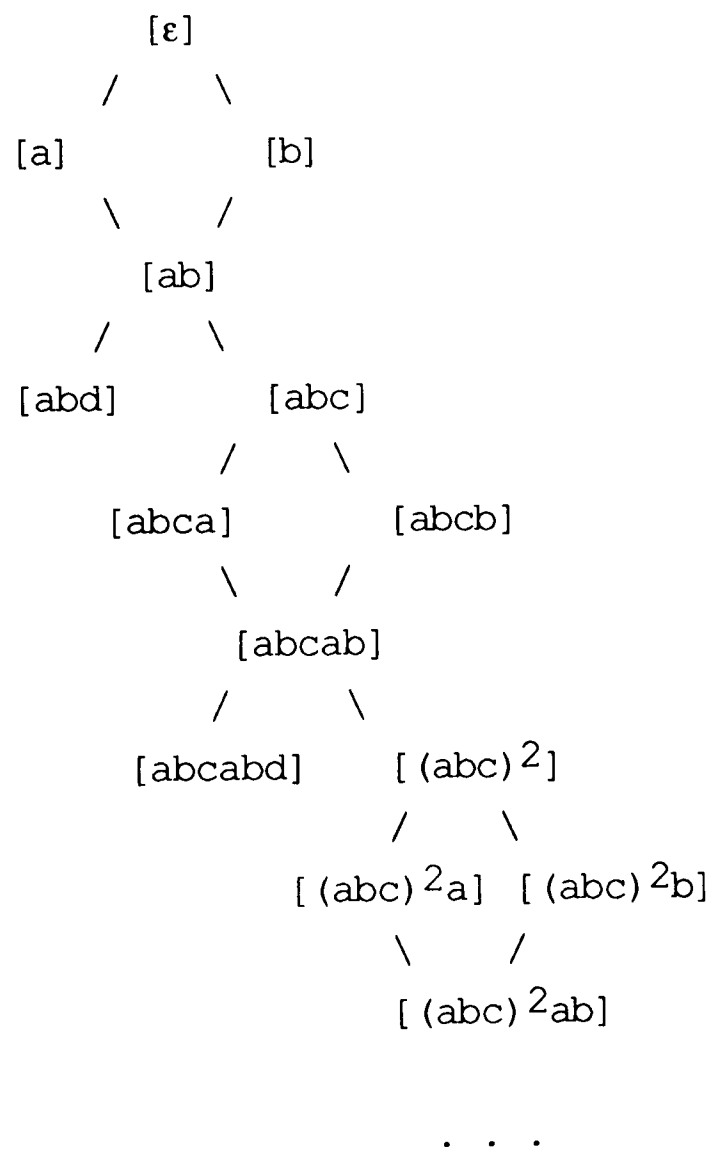


Fig. 3.2.3. The poset of all prefixes of the trace language {[$(abc)^*$abd]}.

Decomposition of Infinitary Trace Languages

Let $T \subseteq \Theta_\iota{}^\infty$. Then *decomposition* of T, denoted $\Delta(T)$, is given by:

$$\Delta(T) = \cup \{\Delta(\tau) \mid \tau \in T\}.$$

Proposition 3.2.12. The following properties of decomposition of trace languages extend onto infinitary languages.

(i) $T = \Delta(T)$ if $\iota = \varnothing$

(ii) $\Delta(\Delta(T)) = \Delta(T)$.

The language T is *connected* if, and only if, $T = \Delta(T)$.

3.3. Asynchronous Transition Systems

Labelled transition systems are often used to provide a natural framework for defining operational semantics. Although they were originally introduced as models for parallel computations [Kel76], we shall dispute this case. We use transition systems extended with the notion of concurrency, which is given in the form of an independency relation over actions within the system. Such an extension is particularly useful when representing asynchronous behaviours, hence the origin of the term *asynchronous transition systems*.

Asynchronous transition systems were developed in [Shi85a] [Shi88c] [Bed87] [Kwi88a]. We shall restrict ourselves to the use of labelled transition systems without ambiguity. This does not greatly restrict the expressive power of the model, as it is usually possible to disambiguate by deriving a suitable labelling of actions. A different approach to ambiguity was presented in [Kwi88a].

Basic Definitions

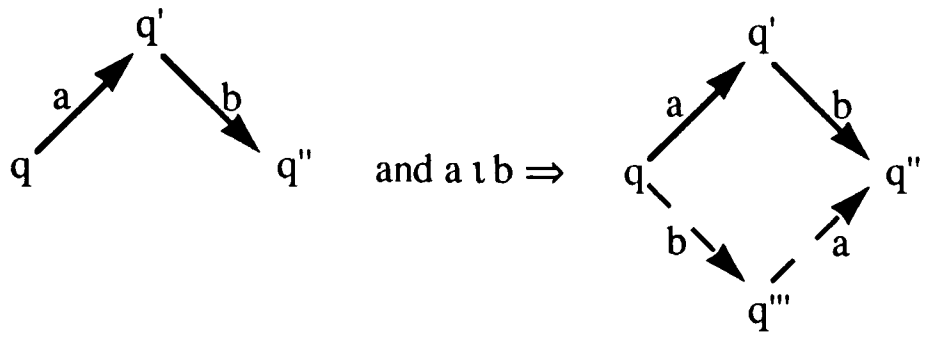
Asynchronous transition systems (ATS) constitute abstract representations of (discrete) systems. They view a system as consisting of indivisible *agents* that can engage in *actions* (possibly shared between agents), the occurrences of which can be monitored by an external observer. Actions that are not shared are *independent* and can happen concurrently; on the other hand, those actions that belong to the same agent are *causally dependent* and must be ordered in time, i.e. sequential, although, at each step, the agent may offer a number of actions as non-deterministic choices.

Definition. (Asynchronous Transition System)
An *asynchronous transition system* (ATS) is a quadruple $(Q, A, \rightarrow, \iota)$ such that:

$(Q, A, \rightarrow)$ is an unambiguous transition system;

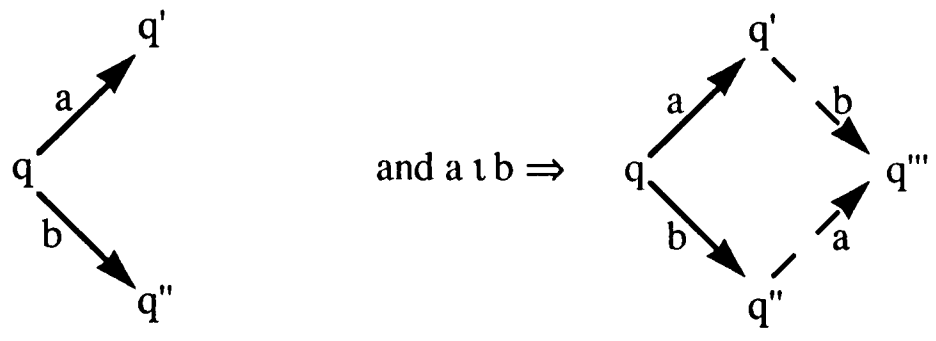
$\iota \subseteq A \times A$ is a causal *independency* of actions (i.e. irreflexive and symmetric relation);

(i) $\forall\ q, q', q'' \in Q, \forall\ a, b \in A$:



An asynchronous transition system is called *forward stable* [Bed87] iff, in addition, the following holds:

(ii) $\forall\ q, q', q'' \in Q, \forall\ a, b \in A$:



Note that we do not restrict the set of states Q to a finite set; however, the set of actions A will be finite.

Conditions (i), (ii) are independent and are also known as the 'diamond property' [Bed87] [Shi88c] [BoC88] [MOP88]. Condition (i) states that any two consecutive independent actions may be permuted and this would have no effect on the result (i.e. the resulting state). Condition (ii) is a conditional Church-Rosser property - it enforces confluence in the context of two independent actions being simultaneously enabled (uniqueness of the resulting state follows from unambiguity).

From now on the concurrent alphabet (A, ι) will remain fixed unless otherwise stated.

Let $S = (Q, A, \rightarrow, \iota)$ be an ATS. A *rooted* asynchronous transition system is an ordered pair $\Sigma = (S, q_0)$, where $q_0 \in Q$.

Example. Let us consider a process p = abNIL || cNIL. Obviously, a ι c and b ι c. Fig. 3.3.1 includes the (rooted) asynchronous transition system that represents it.

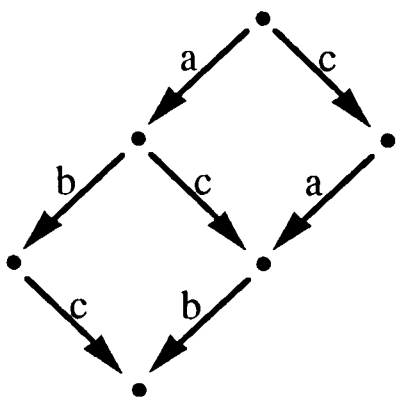


Fig. 3.3.1. ATS determined by p = abNIL || cNIL.

It is easy to verify that the above is a forward stable ATS. On the other hand, the process p' = abcNIL + cabNIL + acbNIL, although characterized by the same sequentializations, gives rise to a different ATS (see Fig. 3.3.2). Here, $\iota = \varnothing$.

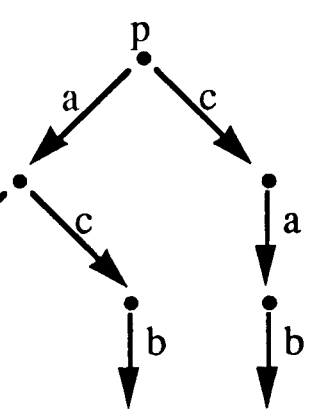


Fig. 3.3.2. ATS determined by p = abcNIL+acbNIL+cabNIL.

Note that there is no confluence in this case. A yet another example would be p" = (fix X.aX) || bNIL, where a ι b. Fig. 3.3.3 shows the corresponding ATS.

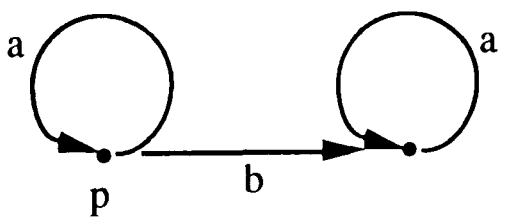


Fig. 3.3.3. ATS determined by p = (fix X.aX) || bNIL.

Fundamental Situations in ATS

We shall now formally define sequence, non-determinism and concurrency in asynchronous transition systems.

Definition. (Sequence, non-determinism, concurrency)
Let $S = (Q, A, \rightarrow, \iota)$ be an ATS. Let $a, b \in A$, $q, q', q'', q''' \in Q$.

(i) We say that actions $a, b \in A$ are *in sequence* at q iff
$$(\exists\, q', q'' \in Q: q \rightarrow^a q' \wedge q' \rightarrow^b q'') \wedge (\neg\, \exists\, q''': q \rightarrow^b q''').$$

(ii) We say $a, b \in A$, $a \neq b$, are *non-deterministic* (*in conflict*) at q iff
$$q \rightarrow^a \wedge\ q \rightarrow^b \wedge\ \neg\, a\, \iota\, b.$$

(iii) We say $a, b \in A$ are *concurrent* at q iff
$$(\exists\, q', q'' \in Q: q \rightarrow^a q' \wedge q' \rightarrow^b q'') \wedge a\, \iota\, b.$$

The presence of concurrency and non-determinism may give rise to a yet another situation called *confusion*, a phenomenon discovered in net theory [RoT86] (and not formulatable in process algebras).

Definition. (Confusion)
Let $S = (Q, A, \rightarrow, \iota)$ be an ATS.

(i) Let $a \in A$. The *conflict set* of a (at q)
$$\mathit{cfl}\ (a, q)$$
is the set
$$\{b \in A \mid (b \neq a) \wedge (q \rightarrow^b) \wedge \neg(a\, \iota\, b)\}.$$

(ii) Let $a, b \in A$, such that $q \rightarrow^a q'$ and $q \rightarrow^b q''$ with $a\, \iota\, b$.
The ordered triple (q, a, b) is a *confusion* (at q) iff
$$\mathit{cfl}\ (a, q) \neq \mathit{cfl}\ (a, q'').$$

(iii) Let $\alpha = (q, a, b)$ be a confusion at q. We say α is a *symmetric* confusion iff
(q, b, a) is also a confusion at q;
otherwise α is *asymmetric*.

Confusion occurs in a system between two independent and simultaneously enabled transitions when conflict sets of one transition are changed through the occurrence of the other one. In other words, conflicts are either resolved or new ones are introduced. Such a situation creates difficulty because sequentializations of the same behaviour can differ radically in terms of non-deterministic choices available at each step.

Confusion is not inherent to the presence of both concurrency and non-determinism in a concurrent system, but, unfortunately, it is not always avoidable.

Example. Let us consider a simple example of confusion. The process p = ((aNIL+bNIL) || (cNIL+$\overline{b}$NIL))\b can either make independent moves a and c, or choose to synchronise on b, which would result in a silent move τ (denoted τ_b). Obviously, a ι c, but a is dependent on τ_b and τ_b is dependent on c. Fig. 3.3.4 shows the ATS determined by p. Note that state 1 corresponds to p, 2 corresponds to (NIL || (cNIL+$\overline{b}$NIL))\b etc. We have:

$cfl(1, a) = \{\tau_b\}$ $\quad$ $cfl(1, c) = \{\tau_b\}$

$cfl(3, a) = \varnothing$ $\quad$ $cfl(2, c) = \varnothing$

thus $\alpha = (1, a, c)$ is a symmetric confusion.

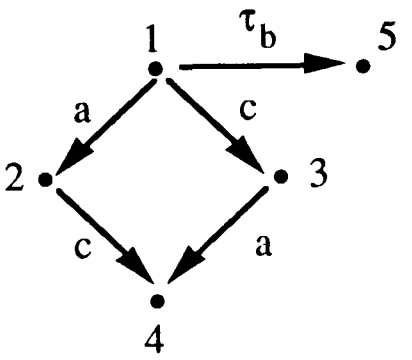


Fig. 3.3.4. ATS that exhibits symmetric confusion (1, a, c), where a ι c.

Hierarchy of Asynchronous Transition Systems

Let S = (Q, A, $\rightarrow$, ι) be an ATS. The following are subclasses of asynchronous transition systems:

Definition. An asynchronous transition system is called *determinate* iff

$$(q \rightarrow^a q' \ \wedge \ q \rightarrow^b q'' \ \wedge \ q' \neq q'') \ \Rightarrow \ a \iota b.$$

for all $q, q', q'' \in Q$, $a, b \in A$: Otherwise it is *non-determinate*.

Determinate systems, also called conflict-free, allow concurrency but disallow non-determinism.

Definition. An asynchronous transition system is called *non-sequential* iff

$$(q \rightarrow^a q' \rightarrow^b q'') \ \wedge \ (a \iota b)$$

for some $q, q', q'' \in Q$, $a, b \in A$. Otherwise it is *sequential*.

Note that transition systems are sequential.

Definition. An asynchronous transition system is called *confusion-free* iff

$\alpha = (q, a, b)$ is not a confusion at q

for all $q \in Q$, $a, b \in A$.

Confusion-free systems form a proper subclass of asynchronous transition systems.

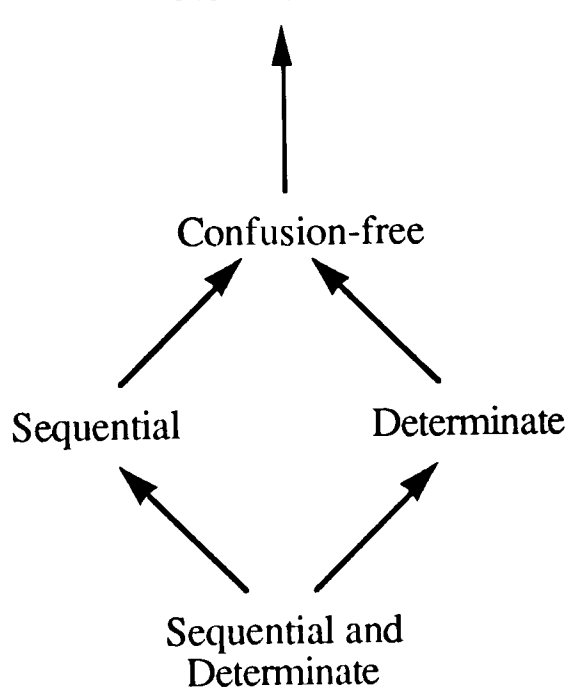


Fig. 3.3.5. Hierarchy of (unambiguous) asynchronous transition systems.

Alphabet Structures

We have so far avoided the question of how to distinguish agents within the system. Let C = (A, ι) be a concurrent alphabet. $\beta \subseteq A$ is an *alphabet* of an agent (process) if, and only if, β is non-empty and:

$$\forall\, a, b \in A:\ \ a\, \iota\, b \Rightarrow \{a, b\} \not\subset \beta.$$

An *alphabet structure* over C is a subset α of the powerset of the set of action labels $\wp(A)$ that satisfies the following conditions:

(i) $\forall\, a, b \in A:\ \neg(a\, \iota\, b) \Rightarrow (\exists\, \alpha_i \in \alpha:\ \{a, b\} \subseteq \alpha_i)$

(ii) $\forall\, \alpha_i \in \alpha:\ \alpha_i$ is an alphabet

(iii) $\forall\, \beta \subseteq A, \forall\, \alpha_i \in \alpha:$ (β alphabet $\wedge\ \alpha_i \subseteq \beta) \Rightarrow (\beta \in \alpha)$.

Let α be an alphabet structure over C. Each $\alpha_i \in \alpha$ corresponds to the alphabet of the agent i. Note that since ι is irreflexive, condition (i) implies that α is a cover of A, that is every action must belong to some agent. Condition (ii) guarantees that no agent in the system can engage in two independent actions, which is a different way of expressing the constraint that all agents are *sequential*, but *asynchronous*. If some action a belongs to the alphabets of more than one agent, then all agents must jointly synchronise whenever this action is enabled in the system. Condition (iii) guarantees that all alphabet structures are in their canonical form only which specifies the same conflicts. For example when A = {a,b,c} and $\iota = \varnothing$ then {{a,b}, {b,c}, {a,c}} is not an alphabet structure, but its canonical form {{a,b,c}, {a,b}, {b,c}, {a,c}} is.

In general, a given concurrent alphabet C = (A, ι) determines a variety of alphabet structures. If α, α' are alphabet structures, so is $\alpha \cup \alpha'$ and $\alpha \cap \alpha'$. Also, α, α' are non-empty.

We can classify alphabet structures depending on the degree of coupling between agents. An alphabet structure $\alpha = \{\alpha_1, \alpha_2, ..., \alpha_k\}$ is *non-communicating* iff $\alpha_i \cap \alpha_j = \varnothing$ for all $1 \le i, j \le k$ distinct. α is *n-way communicating* iff max {card(β) | $\beta \subseteq \alpha$ such that $\bigcap\{\beta_i \mid \beta_i \in \beta\} \neq \varnothing\} = n$. A 2-way communicating alphabet structure is also called *pairwise communicating*.

$a \in \alpha_i$ is a *communication action* of α_i iff $\exists\, \alpha_j \neq \alpha_i$: $a \in \alpha_i \cap \alpha_j$. $a \in \alpha_i$ is a *local action* of α_i iff $\forall\, \alpha_j \neq \alpha_i$: $a \notin \alpha_i \cap \alpha_j$.

Example. Let $A = \{a, b\}$. For $\iota = \{(a,b), (b,a)\}$ then $\{\{a\}, \{b\}\}$ is the only alphabet structure over (A, ι). For $\iota' = \varnothing$ there are four distinct alphabet structures over (A, ι'), the first of which is non-communicating, while the remaining three are pairwise communicating:

$\{\{a,b\}\}$
$\{\{a,b\}, \{a\}\}$
$\{\{a,b\}, \{b\}\}$
$\{\{a,b\}, \{a\}, \{b\}\}$.

Alphabet structures describe the variety of possible decompositions of a given system into asynchronous communicating agents.

Process Structure over ATS

Let $S = (Q, A, \rightarrow, \iota)$ be an ATS. A *process structure* over S is any ordered pair $\Pi = (\alpha, \pi)$ such that:

(i) α is an alphabet structure over (A, ι)

(ii) π is a labelling function ($\pi: Q \rightarrow \wp(\alpha)$) defined by:

$$\pi(q) = \{\alpha_i \in \alpha \mid \exists\, a \in \alpha_i : q \rightarrow^a \}.$$

Let $\Sigma = (S, q_0)$, $q_0 \in Q$, be a rooted ATS. $\Pi = (\alpha, \pi)$ is a process structure over Σ if it is a process structure over S.

Example. Let us consider process $q = (\text{fix } X.(aX+bNIL) \parallel \bar{b}NIL)\backslash b$. The ATS determined by process q is shown in Fig. 3.3.6.

For the alphabet structure $\alpha = \{\{a, \tau_b\}\}$, we have:

$$\pi(q) = \{\{a, \tau_b\}\} \qquad \pi(q') = \varnothing$$

while for $\alpha' = \{\{a, \tau_b\}, \{\tau_b\}\}$:

$$\pi(q) = \{\{a, \tau_b\}, \{\tau_b\}\} \qquad \pi(q') = \varnothing.$$

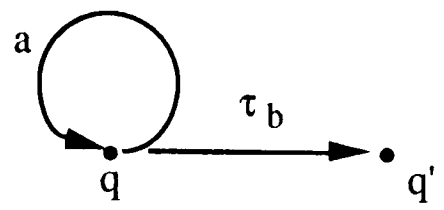


Fig. 3.3.6. ATS determined by q = (fix X.(aX+bNIL) || $\overline{b}$NIL)\b.

A process structure $\Pi = (\alpha, \pi)$ is non-communicating (pairwise communicating, n-way communicating) iff α is non-communicating (pairwise communicating, n-way communicating).

We have simplified the process structure by not distinguishing between two distinct processes with the same alphabet. This means that only static process structures are allowed. Dynamic process structures can be achieved by mapping states into process names[1]. For example, in the infinite ATS shown in Fig. 3.3.7, where a ι b, {a} is the alphabet of processes 1, 11, etc., and {b} is the alphabet of proceses 2, 22, etc. Having executed its transition, each process disappears, and another process with the same alphabet takes over.

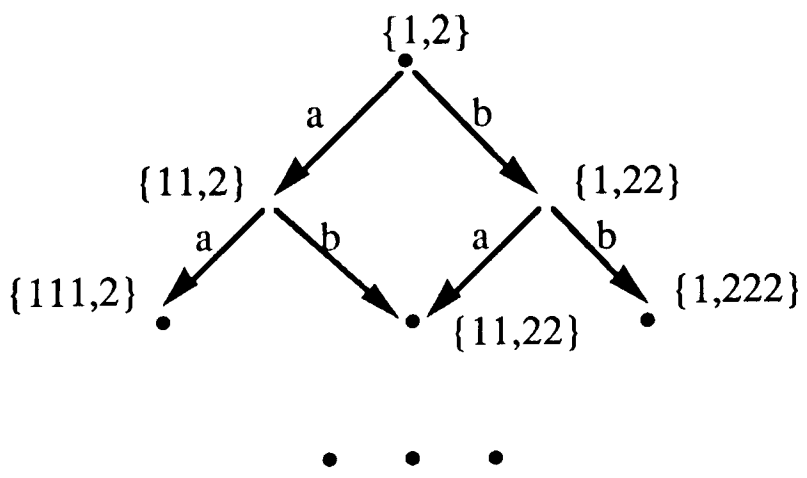


Fig. 3.3.7. An ATS with dynamic processes.

1 We would require an indexed cover [Shi88c] in the more general case.

Order on Process Structures

For a given ATS, any process structure over it is determined by some alphabet structure. We can thus introduce an order on process structures simply by ordering alphabet structures.

Let $C = (A, \iota)$ be a concurrent alphabet, α and α' be alphabet structures over C. We say that α' is *a refinement of* α (denoted $\alpha \leq \alpha'$) if, and only if, $\alpha \subseteq \alpha'$. The refinement relation $\leq$ is a partial order. For a given concurrent alphabet $C = (A, \iota)$ there exist unique *minimal* and *maximal alphabet structures* (denoted α^{min}, α^{max} respectively) with respect to this ordering. The maximal alphabet structure can be constructed as follows:

$$\alpha^{max} = \{\beta \subseteq \wp(A) \mid \forall\, a, b \in \beta:\ (a, b) \notin \iota\}.$$

Note that the maximal alphabet structure must contain singleton sets $\{a\}$, for $a \in A$. The minimal alphabet structure is the alphabet structure α^{min} that satisfies the following condition:

$$\forall\, \alpha_i \in \alpha^{min}, \forall\, \beta \subseteq \wp(A), \beta \neq \alpha_i:\ \beta \subseteq \alpha_i \Rightarrow \beta \notin \alpha^{min}.$$

The set of alphabet structures over a given concurrent alphabet forms a lattice with inclusion ordering. The following is a summary of results concerning alphabet structures.

Proposition 3.3.1.

(i) $\alpha \leq \alpha'$ implies $\forall\, \alpha_i' \in \alpha':\ \exists\, \alpha_k \in \alpha:\ \alpha_i' \subseteq \alpha_k$.

(ii) The class of all alphabet structures over a given concurrent alphabet forms a complete lattice with inclusion ordering.

Proof.

(i) Follows directly from condition (iii) of the definition of alphabet structure.

(ii) The set of all alphabet structures over a finite alphabet is finite; also note that $\alpha \cup \alpha'$ is the lub of α and α', $\alpha \cap \alpha'$ is the glb of α and α'.

❑

Example. In Fig. 3.3.8, $\{\{a,b\}, \{a,c\}\}$ is the minimal alphabet structure, while $\{\{a,b\}, \{a,c\}, \{a\}, \{b\}, \{c\}\}$ is the maximal.

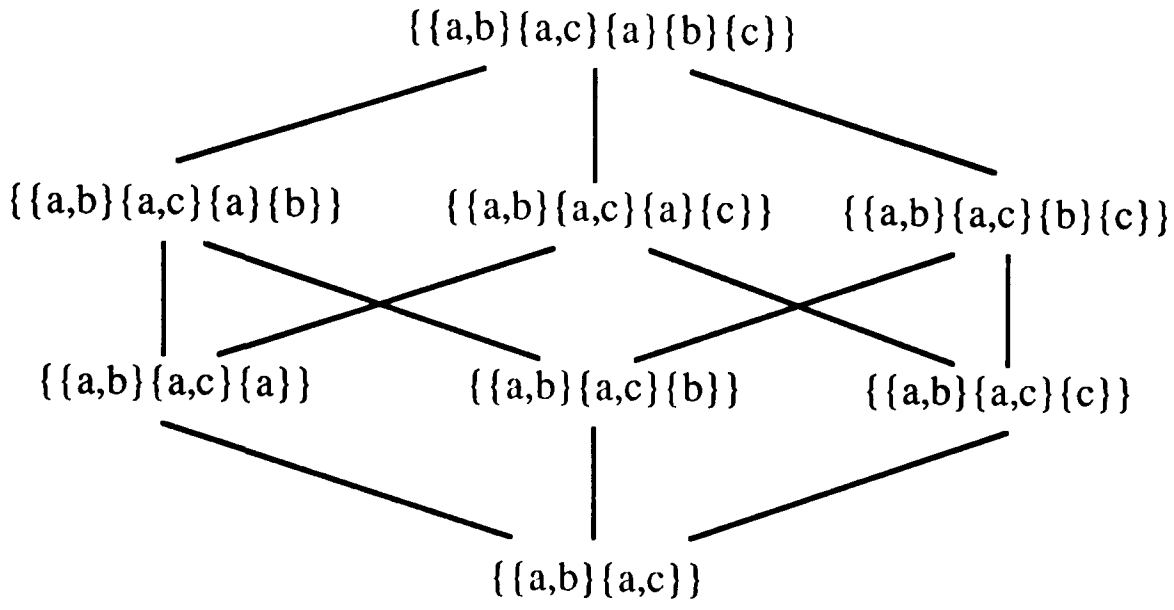


Fig. 3.3.8. The lattice of alphabet structures over A = {a,b,c} with b ι c.

Let S = (Q, A, →, ι) be an ATS, let Σ = (S, q_0), for $q_0 \in Q$, be a rooted ATS, and let Π = (α, π), Π' = (α', π') be process structures over S and Σ . Process structure Π' is *a refinement of* process structure Π (denoted Π ≤ Π') iff α ≤ α'. Π is the *minimal* (*maximal*) process structure over S, Σ iff $\alpha = \alpha^{min}$ (α^{max} respectively).

3.4. Computations in ATS

We now present behavioural representations of asynchronous transition systems. For an asynchronous transition system S = (Q, A, →, ι), we proceed by defining interleaving semantics as a set of derivations, that is sequences in A^{∞}, and trace semantics as a set of traces over the concurrent alphabet (A, ι). We then develop vector representation of traces, which relates process structures over a given ATS with trace semantics.

Derivations

Let S = (Q, A, →, ι) be an ATS. We now extend the transition relation → to strings. We say that $u \in A^{\infty}$ is *applicable* in the state $q \in Q$, denoted $q \rightarrow^{u}$, iff $u = \varepsilon$ or there exists a sequence of states q_i, $i \in \{0, 1,.. \mid u \mid\}$ such that $q_0 = q$ and:

$$q \rightarrow^{u(1)} q_1 \rightarrow^{u(2)} q_2 \rightarrow^{u(3)} q_3 \ldots$$

The above is called a *derivation* of S from q. Note that an infinite sequence $u \in A^\omega$ is applicable in q iff there exists an infinite sequence of states q_i, $i \in N$, such that the above can be formed. A simple consequence of this definition is that, if every finite prefix of an infinite sequence $u \in A^\omega$ is applicable in q, then u is also applicable in q.

Note that if S is an unambiguous asynchronous transition system, derivations are uniquely determined by a start state $q \in Q$ and a sequence of action labels $u \in A^\infty$. We thus choose to consider derivations as elements of A^∞.

The set of all (finite or infinite) derivations of S, $D^\infty(S)$, is defined by :

$$\{u \in A^\infty \mid \exists\, q \in Q:\ q \rightarrow^u\}.$$

We also define finite derivations of S as $D^*(S) = A^* \cap D^\infty(S)$, and infinite ones as $D^\omega(S) = A^\omega \cap D^\infty(S)$. Similarly, the set of all derivations of S from q, denoted $D^\infty(S, q)$, is given by:

$$\{u \in A^\infty \mid q \rightarrow^u\}.$$

Also, $D^*(S, q) = A^* \cap D^\infty(S, q)$, and $D^\omega(S, q) = A^\omega \cap D^\infty(S, q)$.

Let $S = (Q, A, \rightarrow, \iota)$ be an ATS, let $\Sigma = (S, q_0)$, for $q_0 \in Q$, be a rooted ATS. The set of derivations of Σ, denoted $D^\infty(\Sigma)$, is defined by $D^\infty(S, q_0)$. $D^*(\Sigma)$ and $D^\omega(\Sigma)$ are defined correspondingly.

Obviously, $D^\infty(S, q) \subseteq D^\infty(S) \subseteq A^\infty$ for any $q \in Q$; in particular, we have $D^\infty(\Sigma) \subseteq D^\infty(S) \subseteq A^\infty$.

Clearly, $D^\infty(S)$, $D^\infty(S, q)$ and $D^\infty(\Sigma)$ are prefix closed infinitary string languages (with respect to string prefix ordering) and ideals in A^∞. $D^\infty(S, q)$ and $D^\infty(\Sigma)$ are closed, but $D^\infty(S)$ is not. We shall state the following proposition for rooted ATS only.

Proposition 3.4.1. Let $S = (Q, A, \rightarrow, \iota)$ be an ATS, let $\Sigma = (S, q_0)$, for $q_0 \in Q$, be a rooted ATS. $D^\infty(\Sigma)$ is: prefix closed, ideal, and a closed infinitary string language.

Proof.

1) We show $D^\infty(\Sigma)$ is prefix closed, that is $\mathrm{Pref}(D^\infty(\Sigma)) = D^\infty(\Sigma)$. Obviously,

$D^{\infty}(\Sigma) \subseteq \text{Pref}(D^{\infty}(\Sigma))$ by definition of prefix closure. The converse inclusion follows from the fact that if v is a prefix of some derivation u, that is $q_0 \rightarrow^u$ and $v \leq u$, then $q_0 \rightarrow^v$, and thus v is also a derivation of Σ.

2) Since every prefix closed language is an ideal, it follows by part (1) of this proof that $D^{\infty}(\Sigma)$ is an ideal.

2) We show $D^{\infty}(\Sigma)$ is a closed language, that is $\text{Adh}(D^{\infty}(\Sigma)) \subseteq D^{\infty}(\Sigma)$. Note that $\text{Adh}(L) = \{ y \in A^{\omega} \mid \text{Pref}^{\text{fin}}(y) \subseteq \text{Pref}^{\text{fin}}(L) \}$. Suppose $x \in \text{Adh}(D^{\infty}(\Sigma))$; then, by definition of adherence, $x \in A^{\omega}$ and $\text{Pref}^{\text{fin}}(x) \subseteq D^{*}(\Sigma)$. Thus, from unambiguity, there exists an infinite number of states:

$$q_0 \rightarrow^{x(1)} q_1 \rightarrow^{x(2)} q_2 \rightarrow^{x(3)} q_3 \ldots$$

and we conclude $x \in D^{\omega}(\Sigma)$.

□

Observation 3.4.2. $D^{\infty}(\Sigma)$ is a partial monoid (concatenation of u and w is defined only if u is infinite or else it ends at the state q and w is applicable in q).

Derivations correspond to sequential observations of the system made by local observers capable of noting a single occurrence of an action at a time. The particular ordering of symbols within a string is a result of either concurrency or non-determinism. These two phenomena cannot be distinguished at the level of strings, hence we need to provide richer semantics using additional information given in the form of the independency relation.

Note that, in the interleaving approach, sets of derivations would form a basis of the behavioural model (with added state structure).

Example. The ATS Σ_1 included in Fig. 3.4.1 (a ι b) determines the set of derivations $D^{\infty}(\Sigma_1) = \{\varepsilon, a, b, ab, ba, abc, bac\}$. The ATS Σ_2 shown in Fig. 3.4.2 ($\iota = \varnothing$) determines the same set of derivations.

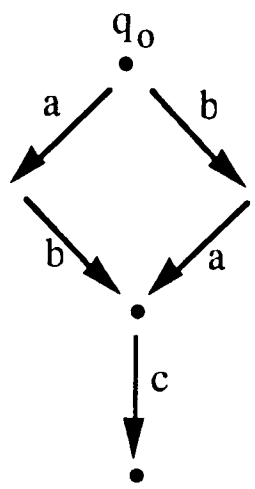


Fig.3.4.1. An asynchronous system Σ_1 determined by q_0 = (aNIL||bNIL);cNIL.

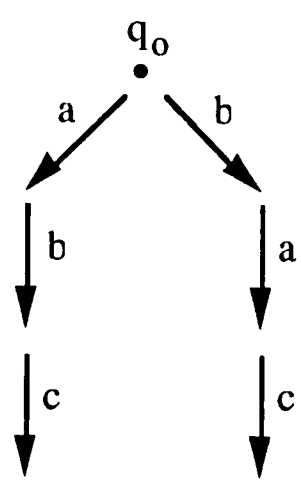


Fig.3.4.2. An asynchronous system Σ_2 determined by q_0 = abcNIL+bacNIL.

Trace Semantics

Let S = (Q, A, $\rightarrow$, ι) be an ATS, and let Σ = (S, q_0), for $q_0 \in$ Q, be a rooted ATS. The set $T^{\infty}(S)$ of *computations* of S is defined by:

$$[\, D^{\infty}(S) \,]_{\iota}$$

that is the set of equivalence classes of derivations with respect to the trace equivalence $\equiv_{\iota}$ determined by independency ι (a trace language). Similarly, $[\, D^{\infty}(S, q) \,]_{\iota}$ and $[\, D^{\infty}(\Sigma) \,]_{\iota}$ define the set $T^{\infty}(S, q)$ of computations of S from q, and the set $T^{\infty}(\Sigma)$ of computations of Σ respectively.

Obviously, $T^{\infty}(S, q) \subseteq T^{\infty}(S) \subseteq \Theta_{\iota}{}^{\infty}$ and $T^{\infty}(\Sigma) \subseteq T^{\infty}(S) \subseteq \Theta_{\iota}{}^{\infty}$ where $\Theta_{\iota}{}^{\infty}$ denotes the set of all traces over the concurrent alphabet (A, ι).

We define $T^{*}(S) = T^{\infty}(S) \cap \Theta_{\iota}{}^{*}$, $T^{*}(S, q) = T^{\infty}(S, q) \cap \Theta_{\iota}{}^{*}$, and $T^{*}(\Sigma) = T^{\infty}(\Sigma) \cap \Theta_{\iota}{}^{*}$. Also, $T^{\omega}(S) = T^{\infty}(S) \cap \Theta_{\iota}{}^{\omega}$, $T^{\omega}(S, q) = T^{\infty}(S, q) \cap \Theta_{\iota}{}^{\omega}$, and $T^{\omega}(\Sigma) = T^{\infty}(\Sigma) \cap \Theta_{\iota}{}^{\omega}$.

The advantage of using traces, rather than derivations, to represent computations is that we can use trace prefix ordering to obtain a partial order relation on computations which abstracts from irrelevant interleavings of concurrent actions.

Example. The ATS Σ_1 (Fig. 3.4.1) is determined by the process (aNIL || bNIL);cNIL, where a ι b. Σ_1 determines the following set of traces:

$$T^{\infty}(\Sigma_1) = \{[\varepsilon], [a], [b], [ab], [abc]\}.$$

Here, the computation [ab] is an extension of both [a] and [b]. On the other hand, the system Σ_2 is determined by the process abcNIL + bacNIL, where $\iota = \varnothing$ (see Fig. 3.4.2). Σ_2 gives rise to the set of traces shown below:

$$T^{\infty}(\Sigma_2) = \{ [\varepsilon], [a], [b], [ab], [ba], [abc], [bac]\}$$

and [ab] is an extension of [a], but not [b]. Note that $D^{\infty}(\Sigma_1) = D^{\infty}(\Sigma_2)$ and string prefix ordering does not view the sequence ab in the system Σ_1 as an extension of the sequence b.

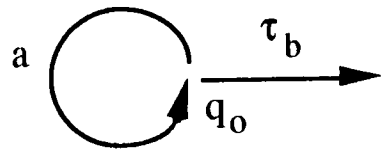


Fig.3.4.3. An asynchronous system Σ_3 determined by:
q_0 = (fix X.(aX+bNIL) || $\overline{b}$NIL)\b.

Example. Let us consider the ATS Σ_3 determined by the process (fix X.(aX+bNIL) || $\overline{b}$NIL)\b, where $\iota = \varnothing$ (see Fig. 3.4.3). The corresponding set of traces is:

$$T^{\infty}(\Sigma_3) = \{[a^*]\} \cup \{[a^*\tau_b]\} \cup \{[a^{\omega}]\}.$$

Admissible Computations

Let $S = (Q, A, \rightarrow, \iota)$ be an ATS, let $\Sigma = (S, q_0)$, for $q_0 \in Q$, be a rooted ATS. A computation $\tau \in T^{\infty}(\Sigma)$ is *maximal* in Σ if it is maximal in $T^{\infty}(\Sigma)$ with respect to trace prefix ordering, that is, there does not exist a computation $\sigma \in T^{\infty}(\Sigma)$ such that $\tau \leq_{\iota} \sigma$ and $\tau \neq \sigma$. The set of all maximal computations of Σ is defined as $\mathrm{Max}(T^{\infty}(\Sigma))$. Maximality of computations in S is defined analogously.

A computation $\tau \in T^{\infty}(\Sigma)$ is *admissible* in Σ iff $\tau \in T^*(\Sigma)$ or τ is maximal in $T^{\infty}(\Sigma)$. A computation $\tau \in T^{\infty}(S)$ is *admissible* in S iff $\tau \in T^*(S)$ or τ is maximal in $T^{\infty}(S)$. The set of all computations admissible in Σ and S is denoted $\mathrm{Adm}(\Sigma)$ and $\mathrm{Adm}(S)$ respectively.

In general, it is not the case that computations maximal in Σ are maximal in $\Theta_{\iota}{}^{\infty}$. Also, note that equivalence classes determined by maximal derivations (with respect to string prefix ordering) are not necessarily maximal in Σ (i.e. with respect to trace prefix ordering). Maximal computations correspond to complete, that is non-extendable, computations of the system. Non-maximal computations correspond to partial, i.e. extendable, computations. In this formalism, infinite, but nevertheless extendable, computations are allowed.

Example. Let us again consider the ATS (denoted Σ) shown in Fig. 3.3.3, where $a \iota b$. Then the following are the sets of derivations and computations determined by Σ:

$$D^{\infty}(\Sigma) = \{a^*\} \cup \{a^*ba^*\} \cup \{a^*ba^{\omega}\} \cup \{a^{\omega}\}$$

$$T^{\infty}(\Sigma) = \{[a^*]\} \cup \{[ba^*]\} \cup \{[ba^{\omega}]\} \cup \{[a^{\omega}]\}.$$

Note that $a^nba^m \equiv ba^{n+m}$, $a^nba^{\omega} \equiv ba^{\omega}$, $[a^{\omega}] \leq [ba^{\omega}]$. Thus, $[a^{\omega}]$ is *not* maximal in $T^{\infty}(\Sigma)$, although a^{ω} is maximal in $D^{\infty}(\Sigma)$. The set of admissible computations of Σ is therefore $T^{\infty}(\Sigma)-\{[a^{\omega}]\}$. The set of maximal computations is $\{[ba^{\omega}]\}$. Intuitively, this corresponds to the process bNIL being only finitely delayed with respect to the concurrent process fix X.aX. Thus, in a maximal computation, the *progress* of action b is guaranteed as long as b is independent of all simultaneously enabled actions that remain in this computation.

Example. Let us consider the ATS Σ_3 included in Fig. 3.4.3. The set of admissible computations of Σ_3 is equal to $T^{\infty}(\Sigma_3) = \{[a^*]\} \cup \{[a^*\tau_b]\} \cup \{[a^{\omega}]\}$. The set of

maximal computations is $\{[a^*\tau_b]\} \cup \{[a^\omega]\}$. Note that in this case, although a and τ_b are simultaneously enabled, the progress of τ_b is *not* guaranteed because a and τ_b are dependent.

A derivation $x \in D^\infty(\Sigma)$ is *admissible* in Σ iff $x \in D^*(\Sigma)$ or [x] is maximal in $T^\infty(\Sigma)$. Admissibility in S is defined analogously.

Example. In the ATS Σ shown in Fig. 3.3.3, where a ι b, the set of derivations admissible in Σ is $D^\infty(\Sigma)-\{a^\omega\}$.

The set of all computations admissible in Σ, $Adm(\Sigma) = T^*(\Sigma) \cup Max(T^\infty(\Sigma))$, is not prefix closed, but it is an ideal, that is, it is closed with respect to finite prefixes: $T^*(\Sigma) = Pref^{fin}(Adm(\Sigma))$. Note that the set of computations maximal in Σ completely characterizes every possible finite partial computation of Σ because $T^*(\Sigma) = Pref^{fin}(Max(T^\infty(\Sigma)))$.

Process Projections

Let $C = (A, \iota)$ be a concurrent alphabet, $B \subseteq A$, B non-empty. For $u \in A^\infty$, the *projection* of u onto B is the sequence u', denoted u/B, in which all symbols that do not belong to B have been deleted.

We now formally define the *projection mapping* /B: $A^\infty \to B^\infty$, for $B \subseteq A$, B non-empty. First, for any $a \in A$, we define the projection of a onto B by:

$$a/B = \begin{cases} \varepsilon & a \in B \\ a & a \notin B. \end{cases}$$

By induction, we can extend the above onto finite strings A^*:

$\varepsilon/B = \varepsilon$,
$(au)/B = (a/B)(u/B)$, for $u \in A^*$, $a \in A$.

Lemma 3.4.3. The projection mapping $/B : A^* \to B^*$ is monotone.

Proof Standard. ❑

The projection mapping can be extended further onto A^∞ by the following definition:

$\forall\, w \in A^\infty$: $w/B = \text{lub}\,\{x/B \mid x \in \text{Pref}^{\text{fin}}(w)\}$.

(Note that this coincides with the inductive definition over A^*). We now show that the projection mapping is well defined and that it is continuous in the domain A^∞ with string prefix ordering.

Proposition 3.4.4. Let $B \subseteq A$.

(i) $\forall\, w \in A^\omega$: w/B exists.

(ii) The projection mapping $/B : A^\infty \to B^\infty$ is continuous.

Proof.

(i) $\text{Pref}^{\text{fin}}(w)$ is a total order, hence it is directed. $\{x/B \mid x \in \text{Pref}^{\text{fin}}(w)\}$ is also totally ordered because the image of a totally ordered set through a monotone mapping is totally ordered. Finally, the least upper bound of this set exists because $(B^\infty, \leq)$ is a cpo.

(ii) $/B$ is monotone on A^* by Lemma 3.4.3. Monotonicity extends easily onto A^∞. We need to show that for every directed set $M \subseteq A^\infty$ and $B \subseteq A$, $(\text{lub}(M))/B = \text{lub}(M/B)$, where M/B denotes $\{x/B \mid x \in M\}$. Suppose $M \subseteq A^\infty$ is directed; then M must be totally ordered by string prefix ordering and lub(M) exists because A^∞ is a cpo (Proposition 3.1.1). Let us denote lub(M) by w. Since $(A^\infty, \leq)$ is algebraic and A^* are the finite elements, $w = \text{lub}\{x \in A^* \mid x \leq w\}$, or, in other words, w is the least upper bound of the set $\text{Pref}^{\text{fin}}(w)$. Let P denote the set $\text{Pref}^{\text{fin}}(w)$. Thus, $w/B = (\text{lub}(P))/B$. If w is finite, then $w \in P$ and the equality of w/B and $\text{lub}(P/B)$ follows trivially. On the other hand, if w is infinite, we have $w/B = \text{lub}(P/B)$ by definition of projection mapping. This concludes the proof since $M \subseteq P$.

□

Let $C = (A, \iota)$ be a concurrent alphabet, $\alpha \subseteq \wp(A)$ be an alphabet structure over C. Let $\alpha = \{\alpha_1, ..., \alpha_n\}$. The *process projection* of a string $u \in A^\infty$ onto a process α_i, also denoted $p_i{}^\alpha(u)$, is defined by u/α_i. Roughly speaking, process projection of a derivation is the sequence of actions that constitute the contribution of the individual agent to the system behaviour. We shall use the projection mapping to localize the semantics of concurrent systems with respect to concurrent agents.

Example. Let A = {a,b,c}, b ι c, α = {{a,b}, {a,c}} where α_1 = {a,b}, α_2 = {a,c}. Then:

(b)/α_1 = b, (b)/α_2 = ε

(abca)/α_1 = aba, (abc)/α_2 = aca.

Vector Semantics

We now introduce a representation of computations of asynchronous transition systems that takes into account process structures. For a given process structure, this is achieved by projecting derivations onto alphabets of processes and taking the corresponding vector of process projections.

Let C = (A, ι) be a concurrent alphabet, $\alpha \subseteq \wp(A)$ be an alphabet structure over C. Let α = $\{\alpha_1, ..., \alpha_n\}$. The mapping p^α: $A^\infty \rightarrow (\alpha_1)^\infty \times (\alpha_1)^\infty \times ... \times (\alpha_n)^\infty$, called the *vectorizing mapping*, is defined by:

$\forall\, u \in A^\infty$: $p^\alpha(u) = \langle p_1{}^\alpha(w), p_2{}^\alpha(w), ...p_n{}^\alpha(w)\rangle$.

For $u \in A^\infty$, $p^\alpha(u)$ is called a *vector representation* of u. Each coordinate of such vector corresponds to one agent and represents a local view of system's behaviour. The superscript α will be omitted if the alphabet structure is understood.

The mapping p^α can be extended onto infinitary languages $L \subseteq A^\infty$ by:

$p^\alpha(L) = \{p^\alpha(u) \mid u \in L\}$.

Let C = (A, ι) be a concurrent alphabet, α be an alphabet structure over C. Then $V^\infty(\alpha)$ denotes the set $p^\alpha(A^\infty)$ of all vectors over a given concurrent alphabet and alphabet structure. Similarly, $V^*(\alpha) = p^\alpha(A^*)$ and $V^\omega(\alpha) = p^\alpha(A^\omega)$. Any subset of $V^\infty(\alpha)$ is called a *vector language* [2] [Shi88c].

Let S = (Q, A, $\rightarrow$, ι) be an ATS, let Σ = (S, q_0), for $q_0 \in Q$, be a rooted ATS, and α be an alphabet structure over (A, ι). We adopt the following notation: $V^\infty(S, \alpha) = p^\alpha(D^\infty(S))$, $V^*(S, \alpha) = p^\alpha(D^*(S))$, and $V^\omega(S, \alpha) = p^\alpha(D^\omega(S))$. Also, $V^\infty(\Sigma, \alpha) = p^\alpha(D^\infty(\Sigma))$, $V^*(\Sigma, \alpha) = p^\alpha(D^*(\Sigma))$, and $V^\omega(\Sigma, \alpha) = p^\alpha(D^\omega(\Sigma))$.

---

2 Shields considers an indexed cover as an alphabet structure.

Example. Let us again consider the ATS Σ_1 shown in Fig. 3.4.1. $\alpha = \{\{a,c\}, \{b,c\}\}$ is a sample alphabet structure over Σ_1. The set of all vectors with respect to α that Σ_1 gives rise to is:

$$V^\infty(\Sigma_1, \alpha) = \{ <\varepsilon,\varepsilon>, <a,\varepsilon>, <\varepsilon,b>, <a,b>, <ac,bc>\}.$$

Example. The ATS Σ_3 shown in Fig. 3.4.3 allows four distinct alphabet structures, for example $\alpha = \{\{a,\tau_b\}\}$, $\alpha' = \{\{a,\tau_b\}, \{\tau_b\}\}$. We can show the following:

$$V^\infty(\Sigma_3, \alpha) = D^\infty(\Sigma_3)$$
$$V^\infty(\Sigma_3, \alpha') = \{<a^*,\varepsilon>\} \cup \{<a^*\tau_b,\tau_b>\} \cup \{<a^\omega,\varepsilon>\}.$$

Clearly, we have:

$$V^\infty(\alpha) \subseteq (\alpha_1)^\infty \times (\alpha_2)^\infty \times \ldots \times (\alpha_n)^\infty$$
$$V^\infty(\Sigma, \alpha) \subseteq V^\infty(S, \alpha) \subseteq V^\infty(\alpha).$$

It should be noted that not every element of $(\alpha_1)^\infty \times (\alpha_2)^\infty \times \ldots \times (\alpha_n)^\infty$ is a valid vector representation of some string in A^∞. For example, if $\alpha = \{\{a,b\}, \{b\}\}$, where a is dependent on b, there is no string x in A^∞ such that $p^\alpha(x) = <a, b>$ (since b is contained in both alphabets, the same number of b's is required in each coordinate). Each coordinate represents some "local history"; for a vector of such local histories to form a consistent global history, all coordinates must agree on the order of shared events.

$V^*(\alpha)$ forms a submonoid of $(\alpha_1)^* \times (\alpha_2)^* \times \ldots \times (\alpha_n)^*$ with coordinatewise concatenation [Maz84a].

Trace Semantics and Vector Representation

We need to show that vector representation of traces is consistent with trace semantics. We first prove that, for $u, v \in A^\infty$, $u \equiv_\iota v$ implies that u and v have the same vector representations, and that this is independent of the alphabet structure α over a given concurrent alphabet (A, ι). We shall make use of the following lemma.

Lemma 3.4.5. Let α be an alphabet structure over (A, ι), $\alpha_i \in \alpha$. Then for all $a, b \in A$:

$$a \,\iota\, b \Rightarrow p_i{}^\alpha(ab) = p_i{}^\alpha(ba),\ p^\alpha(ab) = p^\alpha(ba).$$

Proof.

Standard (by case analysis). Note that, from definition of alphabet structure,whenever $a \iota b$ we have $\forall \alpha_i \in \alpha$: $\{a,b\} \not\subset \alpha_i$.

❑

Proposition 3.4.6.

$\forall u, v \in A^\infty$, α alphabet structure over (A, ι): $u \equiv_\iota v \Rightarrow p^\alpha(u) = p^\alpha(v)$.

Proof.

1) We first prove the finite case. Suppose $u \equiv_\iota v$, for some $u, v \in A^*$, then, from definition of trace equivalence, we have:

$\exists w_0, w_1, \ldots w_n$ $n \geq 0$: $w_0 = u$, $w_n = v$ and

$\forall k$, $1 \leq k \leq n$, $\exists w', w'' \in A^*$, $a,b \in A$: $a \iota b$ and $w_{k-1} = w'abw''$, $w_k = w'baw''$.

Take any w_{k-1}, w_k , and any alphabet structure α, $\alpha_i \in \alpha$. Then from Lemma 3.4.5 $p_i^\alpha(w_{k-1}) = p_i^\alpha(w'abw'') = p_i^\alpha(w')p_i^\alpha(ab)p_i^\alpha(w'') = p_i^\alpha(w')p_i^\alpha(ba)p_i^\alpha(w'') = p_i^\alpha(w_k)$. We have thus shown that $\forall \alpha_i : p_i^\alpha(u) = p_i^\alpha(v)$.

2) We now prove the infinite case. We first show that, for any $u, v \in A^\infty$, $u \leq^\infty v$ implies $\forall \alpha_i \in \alpha$: $p_i^\alpha(u) \leq p_i^\alpha(v)$. Suppose $u \leq^\infty v$ for some $u, v \in A^\infty$. From definition of $\leq^\infty$ we have: $\forall x \in \text{Pref}^{fin}(u)$ $\exists y \in \text{Pref}^{fin}(v)$: $x \leq^* y$, from which, by definition of $\leq^*$, we deduce $\forall x \in \text{Pref}^{fin}(u)$ $\exists y \in \text{Pref}^{fin}(v)$: $\exists z \in A^*$: $x \leq z \wedge z \equiv y$. From monotonicity of p_i^α we have:

2a) $p_i^\alpha(x) \leq p_i^\alpha(z)$ for all $\alpha_i \in \alpha$,

and from part (1) of this proof:

2b) $p_i^\alpha(z) = p_i^\alpha(y)$ for all $\alpha_i \in \alpha$.

The set of finite prefixes x of u forms an increasing chain, hence, by monotonicity of p_i^α, the set $X_i = \{p_i^\alpha(x) \mid x \in \text{Pref}^{fin}(u)\}$ is increasing. Moreover, by (2a) and (2b), every element in this chain is bounded by some element in the increasing chain $Y_i = \{p_i^\alpha(y) \mid y \in \text{Pref}^{fin}(v)\}$. Thus, for all $\alpha_i \in \alpha$, lub $X_i \leq$ lub Y_i and both least upper bounds exist because $(\alpha_i)^\infty$ is a cpo. From continuity of p_i^α, lub $X_i = p_i^\alpha(\text{lub}(\text{Pref}^{fin}(u))) = p_i^\alpha(u)$, and lub $Y_i = p_i^\alpha(\text{lub}(\text{Pref}^{fin}(v))) = p_i^\alpha(v)$. We have thus shown $p_i^\alpha(u) \leq p_i^\alpha(v)$ for all $\alpha_i \in \alpha$.

Suppose $u \equiv_\iota v$ for some $u, v \in A^\infty$; then it follows that $u \leq^\infty v$. Thus $p_i^\alpha(u) \leq p_i^\alpha(v)$

for all $\alpha_i \in \alpha$ as shown above. By symmetry, we have $p_i^{\alpha}(v) \leq p_i^{\alpha}(u)$. This concludes the proof since we have proved $p_i^{\alpha}(u) = p_i^{\alpha}(v)$ for all $\alpha_i \in \alpha$, hence $p^{\alpha}(u) = p^{\alpha}(v)$.

❑

Summarizing, we have shown that if two strings are equivalent with respect to trace equivalence, then, for any alphabet structure, their vector representations are the same. We shall be concerned with the converse of this proposition in the next section.

Projective Equivalence and Preorder

We now introduce the notion of projective equivalence in A^{∞}. Let $C = (A, \iota)$ be a concurrent alphabet, $\alpha \subseteq \wp(A)$ be an alphabet structure over C. Let $\alpha = \{\alpha_1, ..., \alpha_n\}$. Two strings are said to be *projectively equivalent* if, and only if, their vector representations are the same.

Projective equivalence is determined by an alphabet structure, while trace equivalence is determined by independency. We show that projective equivalence coincides with trace equivalence, that is, the choice of alphabet structure does not affect traces as computations of asynchronous transition systems.

Definition. [Shi88c] (Projective equivalence)
$\forall\, u, v \in A^{\infty}$: $u \approx_{\alpha} v \;\Leftrightarrow\; p^{\alpha}(u) = p^{\alpha}(v)$.

We shall need the following lemma.

Lemma 3.4.7. Let $u, w \in A^{*}$, then:

$$p^{\alpha}(u) = p^{\alpha}(v) \Rightarrow \forall\, a \in A: |u|_a = |v|_a,$$

where $|u|_a$ denotes the number of occurrences of the symbol a in u.

Proof. We prove the result by contraposition. Suppose $|u|_a \neq |v|_a$ for some $a \in A$. Take any alphabet structure α over (A, ι); since α is a cover, there exists $\alpha_k \in \alpha$ such that $a \in \alpha_k$. By definition of p_i^{α}, $|p_k^{\alpha}(u)|_a = |u|_a \neq |v|_a = |p_k^{\alpha}(u)|_a$, hence $p^{\alpha}(u) \neq p^{\alpha}(v)$. This concludes the proof.

❑

Lemma 3.4.8.

Let $\alpha \subseteq \wp(A)$ be an alphabet structure over (A, ι), $u, w \in A^*$ s.t. $p^\alpha(u) = p^\alpha(v)$ and:

$$u = waw'$$
$$v = wb_1...b_naw''$$

where $a \neq b_i$ for all i. Then $a \iota b_i$ for all i.

Proof. (By contradiction.)

Suppose:

(i) $p^\alpha(u) = p^\alpha(v)$

(ii) $u = waw'$

$v = wb_1...b_naw''$ where $a \neq b_i$ for all i

(iii) $\exists\, b_j$: $\neg(a \iota b_j)$.

From (iii) we have by definition of alphabet structure that there exists $\alpha_k \in \alpha$ such that $\{a,b_j\} \subseteq \alpha_k$. From (i) it follows by Lemma 3.4.7 that u is a permutation of v, thus w' may be represented as $s'b_js''$ where s' is the shortest such string. Now:

$$p_k(u) = p_k(waw') = p_k(was'b_js'') = p_k(w)ap_k(s')b_jp_k(s'')$$
$$p_k(v) = p_k(wb_1...b_naw'') = p_k(w)p_k(b_1...b_{j-1})b_jp_k(b_{j+1}...b_n)ap_k(w'').$$

Clearly, since $b_1...b_{j-1}$ does not contain a by (ii), it follows that $p_k(u) \neq p_k(v)$. Thus, $p^\alpha(u) \neq p^\alpha(v)$ and we have reached a contradiction with (i).

❑

Lemma 3.4.9. Let $v \in A^\infty$, α be an alphabet structure over (A, ι). Then:

$$p_i^\alpha(\mathrm{Pref}(v)) = \mathrm{Pref}(p_i^\alpha(v))$$

where $p_i^\alpha(X)$ denotes $\{p_i^\alpha(x) \mid x \in X\}$ for $X \subseteq A^\infty$.

Proof. Clear.

❑

Proposition 3.4.10.

Projective equivalence $\approx_\alpha$ coincides with trace equivalence $\equiv_\iota$.

Proof.

1) $u \equiv_\iota v \Rightarrow p^\alpha(u) = p^\alpha(v)$, for any alphabet structure α, has been observed in Proposition 3.4.8.

2) We show $u \approx_\alpha v \Rightarrow u \equiv_\iota v$.

2a) We first prove the finite case. Let α be an alphabet structure over (A, ι). Suppose $u, v \in A^*$ such that $u \approx_\alpha v$. Then $p^\alpha(u) = p^\alpha(v)$ by definition of projective

equivalence. By Lemma 3.4.7, it may be concluded $\forall a \in A$: $|u|_a = |v|_a$, that is v must be a permutation of u. Thus, u and v may be represented by:

$u = waw'$

$v = wb_1...b_naw''$

where $a \neq b_i$ for all i. By Lemma 3.4.8 we have $a \iota b_i$ for all i. We construct:

$v_1 = wab_1...b_nw''$.

Observe that v_1 is a permutation of v obtained through a finite number of permutations $ab_i \rightarrow b_ia$ of two consecutive independent symbols. Thus, $v_1 \equiv_\iota v$ by definition of trace equivalence. Also, since such permutations preserve $\approx_\alpha$ by Lemma 3.4.5, we have $p^\alpha(u) = p^\alpha(v) = p^\alpha(v_1)$. We have thus constructed v_1 such that the length of the common prefix of u and v_1 is increased with respect to the length of the common prefix of u and v. By induction on $|u| - |w|$, where w is the maximal common prefix of u and v_k, with v_k obtained from v at the kth step as described above, we can conclude that $u \equiv_\iota v$.

2b) We now extend the above result onto A^∞ (proof by contradiction). Suppose it is not the case that $u \equiv_\iota v$; then $\neg (u \leq^\infty v)$ or $\neg (v \leq^\infty u)$. We would like to show $\neg (u \leq^\infty v)$ implies $\neg (\forall i: p_i^\alpha(u) \leq p_i^\alpha(v))$. Suppose $\neg (u \leq^\infty v)$; hence by definition of $\leq^\infty$, we have:

$\exists x \in Pref^{fin}(u)\ \forall y \in Pref^{fin}(v): \neg (x \leq^* y)$, which implies, from definition of $\leq^*$:

$\exists x \in Pref^{fin}(u)\ \forall y \in Pref^{fin}(v): \neg (\exists z \in A^*: x \leq z \wedge z \equiv y)$, hence:

$\exists x \in Pref^{fin}(u)\ \forall y \in Pref^{fin}(v): \forall z \in A^*: \neg (x \leq z) \vee \neg (z \equiv y)$, which implies:

$\exists x \in Pref^{fin}(u)\ \forall y \in Pref^{fin}(v): \forall z \in A^*, x \leq z: \neg (z \equiv y)$. From part (1) of this proof we have:

$\exists x \in Pref^{fin}(u)\ \forall y \in Pref^{fin}(v): \forall z \in A^*: x \leq z: \neg (\forall i: p_i^\alpha(z) = p_i^\alpha(y))$, hence

(**) $\exists x \in Pref^{fin}(u)\ \forall y \in Pref^{fin}(v): \forall z \in A^*: x \leq z: \exists i: p_i^\alpha(z) \neq p_i^\alpha(y)$.

Suppose $\forall \alpha_i: p_i^\alpha(u) \leq p_i^\alpha(v)$. This means that each $p_i^\alpha(u)$ is a finite prefix of $p_i^\alpha(v)$ or both $p_i^\alpha(u)$ and $p_i^\alpha(v)$ are infinite, and hence equal. Since p_i^α are continuous (Proposition 3.4.4), we have:

a) $p_i^\alpha(u) = p_i^\alpha(lub\{z \in A^* \mid z \leq u\}) = lub\{p_i^\alpha(z) \mid z \in Pref^{fin}(u)\}$

b) $p_i^\alpha(v) = p_i^\alpha(lub\{y \in A^* \mid y \leq v\}) = lub\{p_i^\alpha(y) \mid y \in Pref^{fin}(v)\}$.

From monotonicity of p_i^α:

$p_i^\alpha(x) \leq p_i^\alpha(z) \leq p_i^\alpha(u)$

$p_i^\alpha(y) \leq p_i^\alpha(v)$.

(a) and (b) and $p_i^{\alpha}(u) \leq p_i^{\alpha}(v)$ imply $lub\{p_i^{\alpha}(z) \mid z \in Pref^{fin}(u)\} \leq lub\{p_i^{\alpha}(y) \mid y \in Pref^{fin}(v)\}$; thus each $p_i^{\alpha}(z)$ must be a finite prefix of $p_i^{\alpha}(v)$. Also, every $p_i^{\alpha}(z)$ must be bounded by some $p_i^{\alpha}(y)$. It follows by Lemma 3.4.9 that for all finite prefixes z of u there exists a finite prefix y of v such that $p_i^{\alpha}(z) = p_i^{\alpha}(y)$.

On the other hand, from (**) we have that for each pair z, y such that $x \leq z$, there exists some k such that $p_k^{\alpha}(y) \neq p_k^{\alpha}(z)$; hence there exists some k such that $p_k^{\alpha}(z)$ is *not* a prefix of $p_k^{\alpha}(v)$. We have thus reached a contradiction, by which we have shown:

$$\neg (\forall i: p_i^{\alpha}(u) \leq p_i^{\alpha}(v)).$$

By symmetry, we consider the case of $\neg (v \leq^{\infty} u)$, which terminates the proof.

❑

The above proposition allows us to conclude that projective equivalence does not depend on the choice of an alphabet structure. Although a variety of alphabet structures over a given concurrent alphabet are possible, and hence vector representations of strings over this alphabet, equivalence classes constitute unique representations of vectors with respect to a given independency.

We can thus extend the vectorizing mapping onto Θ_{ι}^{∞}. The *process projection* mapping $p_i^{\alpha} : \Theta_{\iota}^{\infty} \rightarrow (\alpha_i)^{\infty}$ is defined for $\sigma \in \Theta_{\iota}^{\infty}$ as $p_i^{\alpha}(x)$ for some $x \in \sigma$. Likewise, the *vectorizing mapping* p^{α}: $\Theta_{\iota}^{\infty} \rightarrow V^{\infty}(\alpha)$ is defined for $\sigma \in \Theta_{\iota}^{\infty}$ as $p^{\alpha}(x)$ for some $x \in \sigma$. Since p^{α} must be bijective by Proposition 3.4.10, we shall denote the converse mapping by t_{ι}: $V^{\infty}(\alpha) \rightarrow \Theta_{\iota}^{\infty}$.

Obviously, projective equivalence is determined by projective preorder. The definition now follows.

Definition. (Projective preorder)

$\forall u, v \in A^{\infty}$, alphabet structure α over (A, ι):

$$u \leq_{\alpha} v \iff \forall \alpha_i \in \alpha: p_i^{\alpha}(u) \leq p_i^{\alpha}(v).$$

Observation 3.4.11. $\forall u, v \in A^{\infty}: u \leq_S v \Rightarrow u \leq_{\alpha} v$.

Proof. Clear.

❑

Proposition 3.4.12.

Projective preorder $\leq_\alpha$ coincides with trace preorder $\leq^\infty$.

Proof.

1) $u \leq_\alpha v \Rightarrow u \leq^\infty v$ has been proved in Proposition 3.4.10, where we stated:

$\forall$ i: $p_i{}^\alpha(u) \leq p_i{}^\alpha(v)$ implies $u \leq^\infty v$.

2) The implication $u \leq^\infty v \Rightarrow u \leq_\alpha v$ has been shown in Proposition 3.4.6.

❑

Projective preorder induces a partial order relation on vectors in $A^\infty \times ... \times A^\infty$ defined as follows:

$\underline{w} \leq \underline{w'}$ iff $\forall$ i: $w_i \leq w_i'$.

Since A^∞ is a domain, then $A^\infty \times ... \times A^\infty$ is a product domain with finite elements $A^* \times ... \times A^*$ (a standard result in domain theory [ScG87]). It should, however, be stressed that the set of all vectors $V^\infty(\alpha)$ is not in general equal to $A^\infty \times ... \times A^\infty$.

3.5. Properties of Computation Space

Let $C = (A, \iota)$ be a concurrent alphabet. $\Theta_\iota{}^\infty$ is called the *computation space*. It contains the computations of the class of all asynchronous transition systems over the concurrent alphabet (A, ι). Similarly, A^∞ may be viewed as the computation space of all sequential transition systems over the alphabet A. Thus, $\Theta_\iota{}^\infty$ is a generalization of A^∞ with a notion of concurrency given by the independency relation. One would expect that certain properties of A^∞ also generalize onto $\Theta_\iota{}^\infty$, in particular the fact that A^∞ is a monoid and a domain. We provide a positive answer to this question and also show that $\Theta_\iota{}^\infty$ specializes to A^∞ if the independency is empty.

Order-Theoretic Properties

We now show that $(\Theta_\iota{}^\infty, \leq_\iota)$ is a domain. Note that when $\iota = \varnothing$, $(\Theta_\iota{}^\infty, \leq_\iota)$ is isomorphic with the domain $(A^\infty, \leq)$ with prefix ordering.

The following lemmas are required.

Lemma 3.5.1. Let $\sigma, \gamma \in \Theta_\iota{}^*$, $\beta \in \Theta_\iota{}^\infty$.
If $\sigma, \gamma \leq_\iota \beta$ then there exists $\delta \in \Theta_\iota{}^*$ such that $\delta = \text{lub}\{\sigma, \gamma\}$, and hence:

$$\text{Act}(\delta) = \text{Act}(\sigma) \cup \text{Act}(\gamma).$$

Proof.
Let $\sigma, \gamma \in \Theta_\iota{}^*$, $\beta \in \Theta_\iota{}^\infty$ such that $\sigma, \gamma \leq_\iota \beta$.
1) We show there exists $\beta' \in \Theta_\iota{}^*$ such that $\sigma, \gamma \leq_\iota \beta'$. Let $x \in \sigma$, $y \in \gamma$, $z \in \beta$; then $x \leq^\infty z$ and $y \leq^\infty z$. Since x and y are finite, we have by definition of $\leq^\infty$ that there exist finite prefixes z_1, z_2 of z such that $x \leq^* z_1$ and $y \leq^* z_2$. Let $\beta' = [\ \max\{z_1, z_2\}\ ]_\iota$. It is clear that $\sigma, \gamma \leq_\iota \beta'$ and $\beta' \in \Theta_\iota{}^*$.

2) We can now assume $\sigma, \gamma, \beta \in \Theta_\iota{}^*$ and $\sigma, \gamma \leq_\iota \beta$. It needs to be shown that $\delta = \text{lub}\{\sigma, \gamma\}$ exists (proof by induction). If $\sigma = [\varepsilon]$ then take $\delta = \gamma$. It is clear that $\delta =$

$\text{lub}\{\sigma, \gamma\}$. Suppose $\sigma \neq [\varepsilon]$ and $[a] \leq_l \sigma$ for some $a \in A$. Since $\sigma \leq_l \beta$ we have by transitivity of $\leq_l$ that $[a] \leq_l \beta$.

There are two cases:

2a) $[a] \leq_l \gamma$ and thus $\gamma = [a](\gamma/[a]) \leq \beta = [a](\beta/[a])$, which implies, by left cancellation law, that $\gamma/[a] \leq \beta/[a]$. By symmetry, $\sigma/[a] \leq \beta/[a]$. By induction on the length of σ it may be concluded that $\delta' = \text{lub}\{\sigma/[a], \gamma/[a]\}$ exists. Then the following is the required least upper bound:

$$\delta = \text{lub}\{\sigma, \gamma\} = [a]\delta'.$$

2b) $\neg([a] \leq_l \gamma)$, which implies that $\forall$ i such that $a \in \alpha_i$: $\neg(a \leq p_i(\gamma))$. On the other hand, we have:

$\forall$ i such that $a \in \alpha_i$: $p_i(\sigma) \leq p_i(\beta) = p_i([a](\beta/[a])) = ap_i(\beta/[a])$, and

$\forall$ j such that $a \notin \alpha_j$: $p_j(\sigma) \leq p_j(\beta) = p_j([a](\beta/[a])) = p_j(\beta/[a])$.

Thus, since $p_i(\gamma) \leq p_i(\beta)$, we have $\forall$ i: $p_i(\gamma) \leq p_i(\beta/[a])$, from which it follows $\gamma \leq_l \beta/[a]$. By induction on the length of σ it may be concluded that $\delta' = \text{lub}\{\sigma/[a], \gamma\}$ exists. Finally, the following is the required least upper bound:

$$\delta = \text{lub}\{\sigma, \gamma\} = [a]\delta'$$

The conclusion that $\text{Act}(\delta) = \text{Act}(\sigma) \cup \text{Act}(\gamma)$ follows from construction.

❑

Lemma 3.5.2. Let $(P, \leq)$ be a poset.

(i) $X \subseteq P$ is directed $\Leftrightarrow$ $\downarrow X$ is directed.

(ii) Let $X \subseteq P$ be directed. Then:

$$\text{lub}(X) \text{ exists} \Leftrightarrow \text{lub}(\downarrow X) \text{ exists}$$

and if these conditions are satisfied then:

$$\text{lub}(X) = \text{lub}(\downarrow X).$$

(iii) Let $X \subseteq P$ be a finite directed set. Then lub(X) exists and belongs to X.

Proof.

(i), (ii) Standard results [GHK80].

(iii) Proof by induction, omitted.

❑

Theorem 3.5.3. $(\Theta_l^\infty, \leq_l)$ is a domain. The finite elements are Θ_l^*.

Proof.

1) Clearly, $[\varepsilon]$ is the least element. We prove that every directed subset X of Θ_l^∞ has a least upper bound. By Lemma 3.5.2 it is sufficient to show the existence of least upper

bounds of infinite directed sets $X \subseteq \Theta_\iota^\infty$ such that $X = \mathrm{Pref}(X)$. Let $X \subseteq \Theta_\iota^\infty$ be such a set.

We construct lub(X) as a limit of monotonically increasing sequence in $X \cap \Theta_\iota^*$. Since $X \cap \Theta_\iota^*$ is countable, we can order its elements into a sequence $\{\sigma_k \mid k \in N\}$. We define a monotonically increasing family of directed subsets of $X \cap \Theta_\iota^*$ by:

$$M_0 = \varnothing$$

$$M_k = \begin{cases} M_{k-1} \cup \{\sigma_k\} & \text{if } M_{k-1} \cup \{\sigma_k\} \text{ is directed} \\ M_{k-1} \cup \{\sigma_k, \beta\} & \text{if } M_{k-1} \cup \{\sigma_k\} \text{ is not directed and } \beta \text{ is a bound for } M_{k-1} \cup \{\sigma_k\}. \end{cases}$$

Note that β always exists by the assumption of directedness of X. Also, $X \cap \Theta_\iota^* = \cup \{M_k \mid k \in N\}$. Since each M_k is finite and directed, its least upper bound exists and belongs to M_k by Lemma 3.5.2(iii). Let us denote $\mathrm{lub}(M_k)$ by γ_k, with $\gamma_k \in M_k$. By construction of the family M_k, we have:

$$\gamma_k \leq_\iota \gamma_{k+1} \text{ for all } k \in N.$$

Let $x_k \in \gamma_k$, $x_{k+1} \in \gamma_{k+1}$; then $x_k \leq^\infty x_{k+1}$. Since each $\gamma_k \in \Theta_\iota^*$ we have by Proposition 3.2.6:

$$x_k \leq^* x_{k+1} \text{ for all } k,$$

and by definition of $\leq^*$:

$$\exists\, z_{k+1}: x_k \leq z_{k+1} \wedge x_{k+1} \equiv_\iota z_{k+1},$$

and thus $z_{k+1} \in \gamma_{k+1}$. We have thus shown:

$$\forall\, k\ \forall\, z_k \in \gamma_k\ \exists\, z_{k+1}: z_k \leq z_{k+1}$$

and hence it is possible to construct inductively a monotonic sequence $\{z_k \mid k \in N\}$ starting from γ_1.

Define $x = \mathrm{lub}\{z_k \mid k \in N\}$ (exists because A^∞ is a cpo by Proposition 3.1.1), and finally take $\gamma = [x]_\iota$.

We need to show γ is a bound for X. Let σ be an arbitrary member of X, and let $y \in \sigma$. We show $y \leq^\infty x$. Let $z' \in \mathrm{Pref}^{\mathrm{fin}}(y)$, then $[z'] \in X \cap \Theta_\iota^*$ by the assumption that $X = \mathrm{Pref}(X)$. Thus, by construction of x, $\exists\, z''$: $z' \equiv_\iota z'' \wedge z'' \leq x$; hence, by definition of $\leq^*$ we have shown for an arbitrary finite prefix z' of y:

$$\exists\, z'' \leq x:\ z' \leq^*\ z''.$$

By definition of $\leq^\infty$ it may be concluded $y \leq^\infty x$, and thus $\sigma \leq_\iota \gamma$ by definition of $\leq_\iota$. It is easy to see that γ is the least upper bound of X.

2) We now show that Θ_ι^* are the finite elements. Suppose $\sigma \in \Theta_\iota^*$ and $M \subseteq \Theta_\iota^\infty$ is directed. Thus lub(M) exists as shown in part (1) of this proof. We need to show the existence of $\tau \in M$ such that $\sigma \leq_\iota \tau$ if $\sigma \leq_\iota$ lub(M).

Assume $\sigma \leq_\iota$ lub(M) and lub(M) $\in$ M, then we can take τ = lub(M). Otherwise, if $\sigma \leq_\iota$ lub(M) and lub(M) $\notin$ M, then M must be infinite by Lemma 3.5.1.b(iii); hence, lub(M) must be an infinite trace while σ is its finite prefix, and thus $\sigma \neq$ lub(M). Let γ be an arbitrary member of M, then $\sigma, \gamma \leq_\iota$ lub(M). It follows that $M \cup \{\sigma, \text{lub}(M)\}$ is directed. Since $\sigma \neq$ lub(M), we conclude $M \cup \{\sigma\}$ is directed. Thus, there must exist $\tau \in$ M such that $\sigma, \gamma \leq_\iota \tau$.

In order to show that elements of Θ_ι^ω are not the finite elements, let us take any $\sigma \in \Theta_\iota^\omega$ and directed set $M = \text{Pref}^{\text{fin}}(\sigma)$. Then σ = lub(M) $\notin$ M and for no $\tau \in$ M do we have $\sigma \leq_\iota \tau$.

3) We now show that $(\Theta_\iota^\infty, \leq_\iota)$ is algebraic. Let $\sigma \in \Theta_\iota^\infty$ and define $M_\sigma = \{\tau \in \Theta_\iota^* \mid \tau \leq_\iota \sigma\}$. Clearly, M is non-empty, because it contains $[\varepsilon]$. Let $\gamma_1, \gamma_2 \in M_\sigma$, then $\gamma_1 \leq_\iota \sigma$, $\gamma_2 \leq_\iota \sigma$ by definition of M_σ. Hence, by Lemma 3.5.1, there exists a finite prefix δ of σ such that $\gamma_1 \leq_\iota \delta$, $\gamma_2 \leq_\iota \delta$. Since $\delta \in M_\sigma$ by definition of M_σ, it follows that M_σ is directed. It is easy to see that $\sigma = \text{lub}(M_\sigma)$.

Since Θ_ι^* is countable, it may be concluded $(\Theta_\iota^\infty, \leq_\iota)$ is a domain.

❑

It can also be shown that $(\Theta_\iota^\infty, \leq_\iota)$ is a prime algebraic and coherent poset (proof can be found in [Shi88c]). The set of complete primes is a proper subset of the set of finite elements.

Infinitary Trace Monoid

We now introduce the definition of concatenation in Θ_ι^∞. We would like this definition to be a suitable generalization of concatenation in A^∞, where the concatenation of x and y has been defined as the infinite string x if x is infinite, and the infinite string xy in case x is finite but y infinite. This corresponds to our intuitive understanding of *sequential*

composition of two (sequential) behaviours: if the behaviour represented by x does not terminate, then the behaviour represented by y should never proceed. However, the situation becomes more complex when defining sequential composition of two *non-sequential* behaviours. In particular, we must take causal independency into account when defining concatenation over $\Theta_\iota{}^\infty$. For example, when a ι b, it would be incorrect to define $[a^\omega][b]$ as $[a^\omega]$ because the occurrence of a can in no way affect b; thus, the occurrence of a should not prevent b from proceeding. In trace semantics, this is expressed by the traces $[a^\omega]$ and [b] being consistent, that is having a common dominating trace, e.g. $[ba^\omega]$. On the other hand, if a and b are dependent, $[a^\omega][b]$ should be $[a^\omega]$. Note that in this case the traces $[a^\omega]$ and [b] are inconsistent.

Unfortunately, in contrast to the definition for finite traces, we cannot define concatenation of infinite traces as the equivalence class of the concatenation of any pair of representants. Intuitively, when a ι b, the concatenation of $[a^\omega]$ and $[b^\omega]$ should allow for both a and b to proceed infinitely often because a and b are not causally related. In other words, the equivalence class $[(ab)^\omega]$, which contains a fairmerge of a^ω and b^ω, should be the required result. However, if we attempt to apply the law [x][y] = [xy], which holds over $\Theta_\iota{}^*$, then the result of concatenating strings a^ω and b^ω is a^ω. Note that $[a^\omega]$ is not the same as $[(ab)^\omega]$.

Surprisingly, coordinatewise concatenation of vector representations of traces is not helpful either (it is not closed in $V^\infty(\alpha)$). Let us consider A = {a,b,c} with b ι c. The minimal process structure over (A, ι) is {{a,b}, {a,c}}. Vector representation of $[ab^\omega]$ is $\langle ab^\omega, a\rangle$, [ca] is <a, ca>. The coordinatewise concatenation of $\langle ab^\omega, a\rangle$ and <a, ca> is $\langle ab^\omega, aca\rangle$, which is not a valid vector representation because for no string u do we have $p^\alpha(u) = \langle ab^\omega, aca\rangle$ (since a is in both alphabets, the same number of a's in each coordinate is required). Choosing maximal alphabet structure does not solve the problem.

One might also think that the decomposition of traces would provide a satisfactory solution. However, pairwise concatenation of independent components of two possibly infinite traces does not yield the desired result. For example, if b ι c then $\Delta([ab^\omega]) = \{[ab^\omega]\}$, $\Delta([ca]) = \{[ca]\}$. The result of concatenating $\{[ab^\omega]\}$ with {[ca]} is $\{[ab^\omega]\}$, but the intuition would suggest $[acb^\omega]$ here because the fact that b is repeated indefinitely should not delay c.

However, it is possible to define concatenation in Θ_ι^∞ and show that Θ_ι^∞ is a monoid. The approach here differs from [Shi88c], where the concatenation has been defined so that the class of all vectors forms a partial monoid[1].

First, we introduce the notion of compatibility of traces which is a weakened version of the independency relation over traces. A trace $\tau \in \Theta_\iota^\infty$ is *tail-independent* with the trace $\sigma \in \Theta_\iota^\infty$ if, and only if:

$$\exists\, \sigma' \in \mathrm{Pref}^{\mathrm{fin}}(\sigma),\ \forall\, \sigma'' \in \Theta_\iota^*: \sigma'\sigma'' \leq_\iota \sigma \ \Rightarrow\ \sigma'', \tau \text{ are independent.}$$

We shall also require the following notion. We say the trace $\tau \in \Theta_\iota^\infty$ is *tail-independent* with the trace $\sigma \in \Theta_\iota^\infty$ *after* σ' if, and only if:

$$\sigma' \in \mathrm{Pref}^{\mathrm{fin}}(\sigma) \wedge\ \forall\, \sigma'' \in \Theta_\iota^*: \sigma'\sigma'' \leq_\iota \sigma \ \Rightarrow\ \sigma'', \tau \text{ are independent.}$$

The following are easy observations.

Observation 3.5.4.

(i) For every $\sigma \in \Theta_\iota^\infty$, σ is tail-independent with $[\varepsilon]$.

(ii) For all $\sigma, \tau \in \Theta_\iota^\infty$, if σ, τ independent then σ, τ tail-independent.

(iii) For all $\sigma, \tau \in \Theta_\iota^*$:

if σ, τ tail-independent after σ' then $\sigma, \sigma'\tau$ consistent and $\sigma\tau = \sigma'\tau(\sigma/\sigma')$.

Proof. Clear. ❑

Let us define an auxiliary operator $\Lambda(\sigma,\tau)$ for $\sigma, \tau \in \Theta_\iota^\infty$ as follows. Let Σ denote $\mathrm{Pref}^{\mathrm{fin}}(\sigma)$, T denote $\mathrm{Pref}^{\mathrm{fin}}(\tau)$. Then:

$$\Lambda(\sigma,\tau) = \{\sigma'\tau' \in \Theta_\iota^* \mid \sigma' \in \Sigma \wedge \tau' \in T \wedge \tau' \text{ tail-independent with } \sigma \text{ after } \sigma'\}.$$

Clearly, $\Sigma \subseteq \Lambda(\sigma,\tau)$. Intuitively, we break the "past history" of the behaviour σ into finite segments $\sigma' \leq_\iota \sigma$. Similarly, the "potential future" behaviour τ is broken into finite segments $\tau' \leq_\iota \tau$. We then concatenate the finite past σ' with the finite future τ', and choose only those segments $\sigma'\tau'$ which represent a viable *continuation* of the past history given by σ, in the sense that $\sigma'\tau'$ is consistent with σ. Thus, $\Lambda(\sigma,\tau)$ represents all *finite*

---

1 Shields uses an indexed cover as an alphabet structure.

pieces of the new history. It is clear that for finite traces σ and τ the whole of the trace $\sigma\tau$ becomes the new history; however, when traces σ and τ are infinite, part of the future τ may be delayed indefinitely.

We can now define concatenation in Θ_ι^∞.

Definition. (Concatenation in Θ_ι^∞)

We shall define concatenation over Θ_ι^∞ by:

$$\sigma, \tau \in \Theta_\iota^\infty: \quad \sigma\tau = \text{lub}(\Lambda(\sigma,\tau)).$$

Proposition 3.5.5.

(i) For all $\sigma, \tau \in \Theta_\iota^\infty$, $\sigma\tau$ is well defined.

(ii) If $\sigma, \tau \in \Theta_\iota^*$, then $\Lambda(\sigma,\tau) = \text{Pref}^{\text{fin}}([xy])$, for some $x \in \sigma$, $y \in \tau$.

Proof.

(i) We need to show $\Lambda(\sigma,\tau)$ is a directed set. Let Σ denote $\text{Pref}^{\text{fin}}(\sigma)$, T denote $\text{Pref}^{\text{fin}}(\tau)$. Note that $\Lambda(\sigma,\tau)$ is non-empty because $\Sigma \subseteq \Lambda(\sigma,\tau)$ and Σ contains $[\varepsilon]$. Let $\sigma_1\tau_1, \sigma_2\tau_2 \in \Lambda(\sigma,\tau)$, then by definition

(a) $\sigma_1 \in \Sigma \wedge \tau_1 \in T \wedge \tau_1$ tail-independent with σ after σ_1

(b) $\sigma_2 \in \Sigma \wedge \tau_2 \in T \wedge \tau_2$ tail-independent with σ after σ_2

By Lemma 3.5.1 we have there exists $\sigma' \in \Sigma$ such that $\sigma_1, \sigma_2 \leq_\iota \sigma'$ and $\text{Act}(\sigma') = \text{Act}(\sigma_1) \cup \text{Act}(\sigma_2)$. Thus, by definition, τ_1 must be tail-independent with σ after σ' and τ_2 tail-independent with σ after σ'. We now construct $\tau' \in T$ such that $\tau' = \text{lub}\{\tau_1, \tau_2\}$ as in Lemma 3.5.1. Then, $\text{Act}(\tau') = \text{Act}(\tau_1) \cup \text{Act}(\tau_2)$. Let $\sigma'' \in \Theta_\iota^*$ such that $\sigma'\sigma'' \leq_\iota \sigma$. It is easy to see $\text{Act}(\sigma'') \times \text{Act}(\tau') \subseteq \iota$, hence σ'', τ' are independent. It follows τ' is tail-independent with σ after σ'. Let $\sigma_i\sigma_i'' = \sigma'$ for $i = 1, 2$. Then, since τ_i, σ_i'' are independent:

$$\sigma_i\tau_i \leq_\iota \sigma_i\tau_i\sigma_i'' = \sigma_i\sigma_i''\tau_i = \sigma'\tau_i \leq_\iota \sigma'\tau'.$$

We have thus constructed $\sigma'\tau' \in \Lambda(\sigma,\tau)$ such that $\sigma_1\tau_1, \sigma_2\tau_2 \leq_\iota \sigma'\tau'$.

Since Θ_ι^∞ is a cpo, every directed set has a least upper bound. Thus we have shown that concatenation is well defined because $\Lambda(\sigma,\tau)$ is directed, hence its least upper bound exists.

(ii) Clear.

□

It can be easily observed that if σ is maximal in Θ_ι^∞ then the concatenation of σ with any trace τ is equal to σ.

Proposition 3.5.6. If $\iota = \varnothing$ then for any $\sigma, \gamma \in \Theta_\iota^\infty$ we have:

$$\sigma\gamma = [xy]$$

where $x \in \sigma$, $y \in \tau$.

Proof. When $\iota = \varnothing$, $(\Theta_\iota^\infty, \leq_\iota)$ is isomorphic to $(A^\infty, \leq)$ with string prefix ordering. It is easy to see that $\mathrm{lub}(\Lambda(\sigma,\tau))$ is $\{xy\}$ for $x \in \sigma$, $y \in \tau$. Note that all infinite sequences in A^∞ are maximal, and hence $\mathrm{lub}(\Lambda(\sigma,\tau)) = \{x\}$ if $x \in A^\omega$.

❑

Theorem 3.5.7.
Θ_ι^∞ is a monoid.

Proof.
1) $[\varepsilon]$ is the identity.
Let $\sigma \in \Theta_\iota^\infty$, $\Sigma = \mathrm{Pref}^{\mathrm{fin}}(\sigma)$. Then each $\sigma' \in \Sigma$ is independent of $[\varepsilon]$, hence $[\varepsilon]\sigma = \mathrm{lub}(\Sigma) = \sigma$.

Similarly, $\sigma[\varepsilon] = \sigma$.

2) Associativity.
Let $\sigma, \tau, \gamma \in \Theta_\iota^\infty$. Let Σ, T, Γ denote $\mathrm{Pref}^{\mathrm{fin}}(\sigma)$, $\mathrm{Pref}^{\mathrm{fin}}(\tau)$ and $\mathrm{Pref}^{\mathrm{fin}}(\gamma)$ respectively, and let $\sigma' \in \Sigma$, $\tau' \in$ T, $\gamma' \in \Gamma$. We need to show $(\sigma\tau)\gamma = \sigma(\tau\gamma)$. We have the following from definition of concatenation:

(2a) $(\sigma\tau)\gamma = \mathrm{lub}\Lambda(\sigma\tau,\gamma)$
$= \mathrm{lub}\{\sigma'\tau'\gamma' \mid \sigma'\tau' \in \mathrm{Pref}^{\mathrm{fin}}(\sigma\tau) \wedge \gamma' \in \Gamma$
$\wedge\ \gamma'$ tail-independent with $\sigma\tau$ after $\sigma'\tau'\}$
$= \mathrm{lub}\{\sigma'\tau'\gamma' \mid \sigma'\tau' \in \mathrm{Pref}^{\mathrm{fin}}(\sigma\tau) \wedge \sigma' \in \Sigma \wedge \tau' \in \mathrm{T} \wedge \gamma' \in \Gamma$
$\wedge\ \gamma'$ tail-independent with $\sigma\tau$ after $\sigma'\tau'$
$\wedge\ \tau'$ tail-independent with σ after $\sigma'\}$

(2b) $\sigma(\tau\gamma) = \mathrm{lub}\Lambda(\sigma,\tau\gamma)$
$= \mathrm{lub}\{\sigma'\tau'\gamma' \mid \sigma' \in \Sigma \wedge \tau'\gamma' \in \mathrm{Pref}^{\mathrm{fin}}(\tau\gamma)$
$\wedge\ \tau'\gamma'$ tail-independent with σ after $\sigma'\}$
$= \mathrm{lub}\{\sigma'\tau'\gamma' \mid \tau'\gamma' \in \mathrm{Pref}^{\mathrm{fin}}(\tau\gamma) \wedge \sigma' \in \Sigma \wedge \tau' \in \mathrm{T} \wedge \gamma' \in \Gamma$
$\wedge\ \tau'\gamma'$ tail-independent with σ after σ'
$\wedge\ \gamma'$ tail-independent with τ after $\tau'\}$

From (2a) we have (γ' tail-independent with $\sigma\tau$ after $\sigma'\tau'$) $\wedge$ (τ' tail-independent with σ after σ') $\Leftrightarrow$ (γ' tail-independent with σ after σ') $\wedge$ (γ' tail-independent with τ after τ') $\wedge$ (τ' tail-independent with σ after σ').
From (2b) we have ($\tau'\gamma'$ tail-independent with σ after σ') $\wedge$ (γ' tail-independent with τ after τ') $\Leftrightarrow$ (τ' tail-independent with σ after σ') $\wedge$ (γ' tail-independent with σ after σ') $\wedge$ (γ' tail-independent with τ after τ').
This concludes the proof.

❑

Example. The following are examples of concatenation in Θ_ι^∞. Let a ι b, then:
$[a][b^\omega] = lub([b^*] \cup [ab^*]) = [ab^\omega]$
$[ab^\omega][a] = lub([\varepsilon] \cup [ab^*] \cup [aab^*]) = [aab^\omega]$.
$[a^\omega][b^\omega] = lub([a^*b^*]) = lub\{[(a^*b^*)^\omega]\}) = [(ab)^\omega]$

On the other hand, let $\iota = \varnothing$. Then:
$[a][b^\omega] = [ab^\omega]$
$[a^\omega][b] = lub([a^*]) = [a^\omega]$
$[a^\omega][b^\omega] = lub([a^*]) = [a^\omega]$.

Example. The result of concatenating an infinite trace σ with a trace τ is the trace σ extended with a maximal prefix τ' of τ such that $\sigma'\tau'$, for some finite prefix σ' of σ, is consistent with σ. For example, let b ι c. Then:
$[ab^\omega][ca] = lub([\varepsilon] \cup [ab^*] \cup [ab^*c]) = lub\{[ab^\omega], [acb^\omega]\} = [acb^\omega]$.
$[ab^\omega][ac] = lub([\varepsilon] \cup [ab^*]) = [ab^\omega]$.

Left Cancellation for Infinite Traces

We now show that the usual definition of concatenation may be recovered. Left cancellation operator in Θ_ι^* will be required.

Lemma 3.5.8. (Shields)
Suppose x, y $\in \Theta_\iota^*$, then for z $\in \Theta_\iota^\infty$:

$$(xy)/(z \cap (xy)) \geq_\iota x/(z \cap x)$$

where $u \cap v$, for any u, v $\in \Theta_\iota^\infty$, denotes the maximal common prefix of u and v (glb{u,v}).

Proof. (By induction on length of y).
If $y = [\varepsilon]$, then the inequality holds. Suppose $y = y'[a]$, for some $a \in A$. Then we have by induction:

$$(xy')/(z\cap(xy')) \geq x/(z\cap x)$$

and so we need to show that

$$(xy'[a])/(z\cap(xy'[a])) \geq (xy')/(z\cap(xy').$$

Let $x' = xy'$. We have to show:

$$(x'[a])/(z\cap(x'[a])) \geq x'/(z\cap x').$$

Now, there are two cases:
(a) $z\cap(x'[a]) = (z\cap x')[a]$. Observe that $x/(z\cap x')\ \iota\ [a]$, and hence:

$$x'[a] = (z\cap x')[a](x'/(z\cap x')).$$

Thus $(x'[a])/(z\cap(x'[a])) = x'[a]/((z\cap x')[a]) = x'/(z\cap x')$.

(b) $z\cap(x'[a]) = z\cap x'$. Thus:

$$(x'[a])/(z\cap(x'[a])) = x'[a]/(z\cap x') = (x'/(z\cap x'))[a] \geq x'/(z\cap x').$$

This concludes the proof.

❑

Proposition 3.5.9.

Let $\sigma, \gamma \in \Theta_\iota^\infty$. Then

$$\sigma \leq_\iota \gamma \Leftrightarrow \exists\, \beta \in \Theta_\iota^\infty\colon\ \sigma\beta = \gamma.$$

Also, if $\sigma \leq_\iota \gamma$ then there exists the least β such that $\sigma\beta = \gamma$.

Proof.

1) We show that if $\sigma \leq_\iota \gamma$ then there exists (the least) $\beta \in \Theta_\iota^\infty :\ \sigma\beta = \gamma$

Let $\sigma, \gamma \in \Theta_\iota^\infty$ such that $\sigma \leq_\iota \gamma$; then, by Proposition 3.2.10 we have:

$$\mathrm{Pref}^{\mathrm{fin}}(\sigma) \subseteq \mathrm{Pref}^{\mathrm{fin}}(\gamma).$$

Let us denote $\mathrm{Pref}^{\mathrm{fin}}(\sigma)$ by Σ, $\mathrm{Pref}^{\mathrm{fin}}(\gamma)$ by Γ. Let $\tau \in \Gamma - \Sigma$ and define:

$$M_\tau = \{\sigma' \in \Sigma \mid \sigma' \leq_\iota \tau\}.$$

(1a) We show M_τ is directed and contains least upper bounds of every pair of its elements. Observe that M_τ is non-empty because it contains $[\varepsilon]$. Let $\sigma_1, \sigma_2 \in M_\tau$, then:

$\sigma_1, \sigma_2 \leq_\iota \sigma$ because $\sigma_1, \sigma_2 \in \Sigma$

$\sigma_1, \sigma_2 \leq_\iota \tau$ by definition of M_τ.

By Lemma 3.5.1 we have that $\mathrm{lub}\{\sigma_1, \sigma_2\}$ exists and is bounded by both σ and τ; hence:

$$\text{lub}\{\sigma_1, \sigma_2\} \in \Sigma \cap \text{Pref}^{\text{fin}}(\tau) \subseteq M_\tau$$

and we have shown that M_τ is directed and contains least upper bounds of every pair of its elements.

Define:

$$M = \{\tau/\sigma' \mid \tau \in \Gamma-\Sigma, \sigma' = \text{lub}(M_\tau)\}.$$

Observe that M is well defined because τ and σ' are finite (hence τ/σ' is defined) and σ' always exists by (1a). Also note that $M = \varnothing$ if σ is maximal.

(1b)[2] We now show that M is directed. Let $\beta', \beta' \in M$; then $\beta' = \tau'/\sigma'$, $\beta'' = \tau''/\sigma''$, where $\tau', \tau'' \in \Gamma-\Sigma$, $\sigma' = \text{lub}(M_{\tau'})$, and $\sigma'' = \text{lub}(M_{\tau''})$. We need to show there exists $\beta''' \in M$ such that $\beta', \beta' \leq \beta'''$.

For any $\sigma, \gamma \in \Theta_l^\infty$, let us denote their least upper bound by $\sigma \cup \gamma$ (if it exists), and their greatest lower bound by $\sigma \cap \gamma$. Observe that $\tau' \cup \tau''$ exists because Γ bounded by γ (Lemma 3.5.1). Also, by their definition, $\sigma' = \sigma \cap \tau'$ and $\sigma'' = \sigma \cap \tau''$. We show:

$$(\tau' \cup \tau'')/(\sigma \cap (\tau' \cup \tau'')) \geq_l \tau'/(\sigma \cap \tau').$$

Let $y = (\tau' \cup \tau'')/\sigma$. By Lemma 3.5.8:

$$(\tau' \cup \tau'')/(\sigma \cap (\tau' \cup \tau'')) = (\tau' y)/(\sigma \cap \tau' y) \geq_l \tau'/(\sigma \cap \tau').$$

By symmetry, $(\tau' \cup \tau'')/(\sigma \cap (\tau' \cup \tau'')) \geq_l \tau''/(\sigma \cap \tau'')$. Note that $(\tau' \cup \tau'')/(\sigma \cap (\tau' \cup \tau''))$ $\in M$ because $\tau' \cup \tau'' \in \Gamma-\Sigma$ and $\sigma \cap (\tau' \cup \tau'') = \text{lub}(M_{\tau' \cup \tau''})$.

(1c) Since M is directed and Θ_l^∞ is a cpo by Theorem 3.5.3, its least upper bound exists. Define $\beta = \text{lub}(M)$. It follows from construction that lub(M) is the least element β such that $\sigma\beta = \gamma$. It remains to show that $\sigma\beta = \gamma$. Let Σ denote $\text{Pref}^{\text{fin}}(\sigma)$, γ denote $\text{Pref}^{\text{fin}}(\gamma)$.

We show $\Lambda(\sigma,\beta) \subseteq \Gamma$. Suppose $\sigma'\beta' \in \Lambda(\sigma, \beta)$; then, by definition of Λ and β, $\sigma' \in \Sigma$, $\beta' \in M$, and β' tail-independent with σ after σ'. From definition of M we have $\beta' = \tau/\sigma''$ for some $\tau \in \Gamma-\Sigma$, where $\sigma'' = \text{lub}(M_\tau)$. Let us denote $\text{lub}\{\sigma', \sigma''\}$ by δ ($\delta \in \Sigma$ exists by Lemma 3.5.1). Since β' tail-independent with σ after σ', it follows β' tail-independent with σ after δ. Define $\kappa = \delta\beta'$ and observe that κ is the least upper bound of δ, τ. Finally, note that $\kappa \in \Gamma$ because $\delta, \tau \leq \gamma$ (Lemma 3.5.1).

We show $\Gamma \subseteq \Lambda(\sigma,\beta)$. Note that since $\Sigma \subseteq \Gamma$, it is sufficient to show that $\Gamma-\Sigma \subseteq$

---

2 M.W. Shields has helped with the technicalities of this part of the proof.

$\Lambda(\sigma,\beta)$. Let $\tau \in \Gamma-\Sigma$. Define $\beta' = \tau/\sigma'$ where $\sigma' = \text{lub}(M_\tau)$. By definition of M, $\beta' \in M$, hence $\beta' \in \text{Pref}^{\text{fin}}(\beta)$. We need to show β' is tail-independent with σ after σ'. Let σ'' be such that $\sigma'\sigma'' \leq \sigma$. Then both τ and $\sigma'\sigma''$ are bounded by γ, hence their least upper bound $\delta \in \Gamma$ exists by Lemma 3.5.1. Now:

$$\sigma'\sigma''\beta' = \sigma'\sigma''(\tau/\sigma') = \delta = \sigma'\beta'(\delta/\tau) = \sigma'(\tau/\sigma')(\delta/\tau).$$

Since (τ/σ') commutes with $(\delta/\tau) = \sigma''$, we have that $\beta' = (\tau/\sigma') \iota (\delta/\tau) = \sigma''$, which concludes the proof as $\tau = \sigma'\beta' \in \Lambda(\sigma,\beta)$.

2) We show that if $\exists\, \beta \in \Theta_\iota^\infty$: $\sigma\beta = \gamma$ then $\sigma \leq_\iota \gamma$. Suppose $\exists\, \beta \in \Theta_\iota^\infty$: $\sigma\beta = \gamma$, then, by definition of concatenation, $\text{Pref}^{\text{fin}}(\sigma) \subseteq \text{Pref}^{\text{fin}}(\sigma\beta) = \text{Pref}^{\text{fin}}(\gamma)$. Thus, by Proposition 3.2.10, $\sigma \leq_\iota \sigma\beta = \gamma$.

❑

The following left cancellation operator in Θ_ι^∞ may now be introduced. It allows to extract the *continuation* of a trace σ after its prefix γ has been completed. This operator is well defined by Proposition 3.5.9.

Definition. (Left cancellation in Θ_ι^∞)

Let $C = (A, \iota)$ be a concurrent alphabet. Let $\sigma, \gamma, \beta \in \Theta_\iota^\infty$. We say that β is σ *after* γ (denoted σ/γ), where $\gamma \leq_\iota \sigma$, if, and only if, β is the least trace such that $\gamma\beta = \sigma$.

It is easy to notice that this definition agrees with the definition of left cancellation over Θ_ι^*.

Example. If a ι b, then $[a^\omega]/[a^\omega] = [\varepsilon]$, $[ba^\omega]/[a^\omega] = [b]$, $[ba^\omega]/[b] = [a^\omega]$; $[a^\omega]/[b^\omega]$ is undefined. If a is dependent on b, then $[a^\omega]/[a^\omega] = [\varepsilon]$, $[ba^\omega]/[b] = [a^\omega]$; $[ba^\omega]/[a^\omega]$ and $[a^\omega]/[b^\omega]$ are undefined.

Maximal Computations and Processes

Let $C = (A, \iota)$ be a concurrent alphabet, let $S = (Q, A, \rightarrow, \iota)$ be an ATS, and let $\Sigma = (S, q_0)$, for $q_0 \in Q$, be a rooted ATS. Let $\Pi = (\alpha, \pi)$ be a process structure over S. We now summarize the relationship of maximal computations of a given (rooted) ATS Σ and process structures over Σ. Maximal computations and their finite prefixes characterize all admissible computations of a given system. The following theorem states that a non-

maximal computation, whether finite or infinite, is a computation which is *not yet complete*, that is, there exists an action which can extend it non-trivially. In other words, there must exist a process which has proceeded only a finite number of times and which is ready to engage in an action that can (non-trivially) extend the given computation.

It follows that, in a maximal computation, each process proceeds an infinite number of times *unless* it is prevented from proceeding or the computation is already non-extendable. We say maximal computations are therefore *non-extendable*. It is important to notice that there exist infinite computations which, unlike infinite strings, could be extended in the sense that there exists an infinite computation dominating it.

Theorem 3.5.10. The following statements are equivalent.

(i) $\sigma \in T^{\infty}(\Sigma)$ is *not* maximal in Σ

(ii) $\exists\, \gamma \in T^{\infty}(\Sigma)\colon\ \sigma \leq_l \gamma \wedge \sigma \neq \gamma$.

(iii) $\exists\, a \in A : \sigma[a] \in T^{\infty}(\Sigma) \wedge \sigma[a] \neq \sigma$.

(iv) $\forall\, \alpha, \exists\, \alpha_i \in \alpha, \exists\, a \in \alpha_i :$

$$p_i^{\alpha}(\sigma) \in (\alpha_i)^* \wedge \sigma[a] \in T^{\infty}(\Sigma) \wedge \sigma[a] \neq \sigma.$$

Proof

(i) $\Rightarrow$ (ii) Follows directly from the definition of a maximal computation.

(ii) $\Rightarrow$ (iii) Suppose

$$\exists\, \gamma \in T^{\infty}(\Sigma)\colon\ \sigma \leq_l \gamma \wedge \sigma \neq \gamma$$

Let $B = \{b \in A \mid \sigma[b] \in T^{\infty}(\Sigma)\}$. We first show B is non-empty. It follows from (ii) by Proposition 3.5.9, that $\exists\, \beta, \beta \neq [\varepsilon]\colon\ \sigma\beta = \gamma$. Take any prefix [b] of β of length 1; then $[b] \leq_l \beta$, and thus $\sigma[b] \leq_l \sigma\beta = \gamma$. Since $\gamma \in T^{\infty}(\Sigma)$ we have by Proposition 3.6.2 that $\sigma[b] \in T^{\infty}(\Sigma)$. Thus, B is non-empty.
We now show that there exists $a \in B$ such that $\sigma[a] \neq \sigma$. Suppose for all $b \in B$ we have $\sigma[a] = \sigma$. Thus, for all $\gamma \in T^{\infty}(\Sigma)$: $\neg(\sigma \leq_l \gamma) \vee \sigma = \gamma$, and we have reached a contradiction with (ii).

(iii) $\Rightarrow$ (iv) Suppose:

$\exists\, a \in A : \sigma[a] \in T^{\infty}(\Sigma) \wedge \sigma[a] \neq \sigma.$

Let α be an arbitrary alphabet structure. Since α is a cover, $\exists\, \alpha_i \in \alpha$ such that $a \in \alpha_i$. Suppose $p_i(\sigma)$ is infinite for all $\alpha_i \in \alpha$ such that $a \in \alpha_i$ (we prove this part by contradiction). Then, since $\sigma \leq_l \sigma[a]$ by Proposition 3.5.9, $p_i(\sigma[a])$ is also infinite by monotonicity of p_i. Hence:

$p_i(\sigma) = p_i(\sigma[a])$ for all $\alpha_i \in \alpha$ such that $a \in \alpha_i$.

Also, $p_j(\sigma) = p_j(\sigma[a])$ for all $\alpha_j \in \alpha$ such that $a \notin \alpha_j$ by definition of p_j. Hence, $p^{\alpha}(\sigma) = p^{\alpha}(\sigma[a])$, and, by Proposition 3.4.10, $\sigma = \sigma[a]$, which contradicts (iii). We have thus shown that there exists $\alpha_i \in \alpha$ such that $p_i(\sigma) \in (\alpha_i)^*$.

(iv) $\Rightarrow$ (i) Suppose α is an alphabet structure and $\exists\, \alpha_i \in \alpha, \exists\, a \in \alpha_i$: $p_i^{\alpha}(\sigma) \in (\alpha_i)^* \wedge \sigma[a] \in T^{\infty}(\Sigma) \wedge \sigma[a] \neq \sigma$. Thus $\exists\, \gamma \in T^{\infty}(\Sigma), \gamma = \sigma[a]$, such that $\sigma \leq_l \gamma \wedge \sigma \neq \gamma$. Clearly, σ is not maximal in Σ.

❑

Example. Let us consider the ATS shown in Fig. 3.3.3, where a ι b. The only process structure over this ATS is $\{\{a\},\{b\}\}$. Examples of non-maximal computations are $[a^*]$, which can be extended with a or b, and $[a^{\omega}]$, which can be extended with b yielding $[ba^{\omega}]$. Note that $p_2([a^{\omega}])$ is finite. The only maximal computation is $[ba^{\omega}]$.

Example. Let us consider the ATS shown in Fig. 3.4.3, where a is dependent on b. There are four process structures over this ATS, for example $\{\{a,b\}\}$ and $\{\{a,b\},\{b\}\}$. Non-maximal computations are $[a^*]$, which can be extended with a or b. Maximal computations are $[a^*b]$, which cannot be extended, and $[a^{\omega}]$, which, when extended with a or b yields $[a^{\omega}]$.

Maximal Computations and Finite Delay

Maximal computations exhibit a conditional finite delay property [KaM69], which states that a transition cannot be *permanently enabled* in a computation and be *independent of all the remaining transitions* in this computation; either this transition or some transition dependent on it will be taken.

It should be noted that when dealing with determinate systems, which seemed the intention of [KaM69], dependency of transitions may never refer to the transitions being *at conflict*,

but only *in sequence*. In other words, no two dependent transitions may be simultaneously enabled in a determinate system. Thus, for such systems, finite delay property reduces to the statement originally introduced in [KaM69], that is, that *no transition can be permanently enabled and never taken*. When considering non-determinate systems, however, it is possible for a transition to be indefinitely delayed if some action dependent on it is taken instead.

> **Example.** The (determinate) ATS shown in Fig. 3.3.3 has one maximal computation $[ba^{\omega}]$, in which no action has been indefinitely delayed. The computation $[a^{\omega}]$ is not maximal because b is permanently enabled and never taken. On the other hand, the computation $[a^{\omega}]$ of a (non-determinate) system shown in Fig. 3.3.6 is maximal, although action τ_b appears to be indefinitely delayed. Note that conditional finite delay property still applies in this case since transition a, which is dependent on τ_b, is taken.

The following is a characterization of computations that are maximal in $T^{\infty}(\Sigma)$. We use $[a]\ \iota\ \sigma/\gamma$ to denote that traces [a] and σ/γ are independent. In other words, a is independent of all elements in $Act(\sigma/\gamma)$. σ/γ, for $\gamma \leq_{\iota} \sigma$, denotes σ after γ, that is the computation remaining in σ after γ has been completed.

Theorem 3.5.11.

$\sigma \in T^{\infty}(\Sigma)$ is *not* maximal in Σ if, and only if, the following holds:

$$\exists\, \gamma \in Pref^{fin}(\sigma), \exists\, a \in A:$$
$$\gamma[a] \in T^{*}(\Sigma) \wedge ([a]\ \iota\ \sigma/\gamma).$$

Proof (can also be found in [Shi88c]).

1) Suppose $\sigma \in T^{\infty}(\Sigma)$ is *not* maximal in Σ. Then by Theorem 3.5.10(iv) we have for any alphabet structure, say α^{min}:

$$\exists\, \alpha_i \in \alpha^{min}, \exists\, a \in \alpha_i:$$
$$p_i(\sigma) \in (\alpha_i)^{*} \wedge \sigma[a] \in T^{\infty}(\Sigma) \wedge \sigma[a] \neq \sigma.$$

We need to construct $\gamma \in Pref^{fin}(\sigma)$ such that for some $a \in A$:

$$\gamma[a] \in T^{*}(\Sigma) \wedge ([a]\ \iota\ \sigma/\gamma).$$

Let $x \in \sigma$; then $x \in D^{\infty}(\Sigma)$ by Proposition 3.6.1 and $p_i(x) = p_i(\sigma) \in (\alpha_i)^{*}$ by Proposition 3.4.10. Thus, there exists $k \in N$ such that:

$$q_0 \rightarrow^{x(1)} q_1 \rightarrow^{x(2)} \ldots\ q_{k-1} \rightarrow^{x(k)} q_k \ldots$$

and:

(*) $\forall\, j > k$: $x(j) \neq a$, and
$\forall\, j > k$: $x(j) \notin \alpha_i$.

Let $y \in \sigma[a] \in T^{\infty}(\Sigma)$; then $y \in D^{\infty}(\Sigma)$ by Proposition 3.6.1. Since $\sigma[a] \neq \sigma$, we have that there exists $m \in N$ such that:

$$q_0 \rightarrow^{y(1)} q'_1 \rightarrow^{y(2)} \ldots \rightarrow^{y(m)} q'_m \rightarrow^{a} q'_{m+1} \ldots$$

and:

$$y(m) = a, \ \forall j > m: y(j) \neq a.$$

Since $\sigma \leq_{\iota} \sigma[a]$ by Proposition 3.5.9, we have $Pref^{fin}(\sigma) \subseteq Pref^{fin}(\sigma[a])$. Observe that $(\sigma[a])/\sigma = [a]$. Let y' be the prefix of y of length m and define $\gamma = [y']$. Since $[y'] < [y'a]$, we have γ is a prefix of σ.

Clearly, $\gamma \in T^{*}(\Sigma)$ because $y' \in D^{*}(\Sigma)$. Also, $\gamma[a] \in T^{*}(\Sigma)$ because $y'a \in D^{*}(\Sigma)$ and $\gamma[a] = [y'][a]$.

Observe that $Act(\sigma/\gamma) = Act(x/x')$ where $x' \equiv_{\iota} y'$ (x' exists by Proposition 3.6.1 and $x' \in \gamma$), and $|x'| \geq k$. It follows from (*) that for all $b \in Act(\sigma/\gamma) = Act(x/x')$, $b \notin \alpha_i$. Since α^{min} is the minimal alphabet structure, we have $a \in \alpha_i$ and $b \notin \alpha_i$ implies $a \, \iota \, b$. This concludes the proof.

2) Suppose $\exists \gamma \in Pref^{fin}(\sigma), \exists a \in A: \gamma[a] \in T^{*}(\Sigma) \wedge ([a] \, \iota \, \sigma/\gamma)$. Let α^{min} denote the minimal alphabet structure over (A, ι). Since α^{min} is a cover, there exists $\alpha_i \in \alpha^{min}$ such that $a \in \alpha_i$. Since $\gamma(\sigma/\gamma) = \sigma$ and γ finite, it follows $p_i^{\alpha}(\sigma) = p_i^{\alpha}(\gamma(\sigma/\gamma)) = p_i^{\alpha}(\gamma)p_i^{\alpha}(\sigma/\gamma) = p_i^{\alpha}(\gamma)$, because for all $b \in Act(\sigma/\gamma)$ we have $a \, \iota \, b$, and thus $b \notin \alpha_i$. Note that $p_i^{\alpha}(\sigma)$ is finite. We can now show that for every finite prefix γ' of σ such that $\gamma \leq_{\iota} \gamma'$, if γ' can be extended with a and γ' can be extended with b and $a \, \iota \, b$, then $\gamma'[b]$ can be extended with a. Since $\gamma[a] \in T^{*}(\Sigma)$ and $\gamma[b] \in T^{*}(\Sigma)$ for some $b \in Act(\sigma/\gamma)$ and $a \, \iota \, b$, we have $\gamma[a][b] \in T^{*}(\Sigma)$ by condition (ii) of the definition of an asynchronous transition system. By condition (i) of the same definition, $\gamma[b][a] \in T^{*}(\Sigma)$, or, in other words, $\gamma[b]$ must be extendable with a. This allows us to construct a computation $\sigma[a]$ such that $\sigma \leq \sigma[a]$ and $\sigma[a] \in T^{\infty}(\Sigma)$. Clearly, $p_i^{\alpha}(\sigma[a]) = p_i^{\alpha}(\gamma)a$, hence $\sigma[a] \neq \sigma$ because γ finite. Thus, we have shown $\exists \alpha_i \in \alpha^{min}, \exists a \in \alpha_i : p_i^{\alpha}(\sigma) \in (\alpha_i)^{*} \wedge \sigma[a] \in T^{\infty}(\Sigma) \wedge \sigma[a] \neq \sigma$. We therefore conclude $\sigma \in T^{\infty}(\Sigma)$ is *non*-maximal in Σ.

❑

Example. Let $a \, \iota \, b$; then $[a^{\omega}]$, $[ba^{\omega}]$ are not maximal in Θ_{ι}^{∞} because there exist finite prefixes $[\varepsilon]$ and $[b]$ respectively such that $[b]$ remains independent of all $[a^{\omega}]/[\varepsilon] = [a^{\omega}]$

and $[ba^{\omega}]/[b] = [a^{\omega}]$. On the other hand, when a is dependent on b, both $[a^{\omega}]$ and $[ba^{\omega}]$ are maximal.

The following is a consequence of the above theorem. Note that it does not hold for an arbitrary alphabet structure.

Proposition 3.5.12.

$\sigma \in T^{\infty}(\Sigma)$ is *not* maximal in Σ if, and only if, the following holds:

$$\exists\, \alpha_i \in \alpha^{min}, \exists\, a \in \alpha_i :$$

$$p_i^{\alpha}(\sigma) \in (\alpha_i)^* \wedge \sigma[a] \in T^{\infty}(\Sigma).$$

Proof.

This is a direct consequence of the above theorem.

❑

3.6. Relationship of ATS and Trace Languages

Let $S = (Q, A, \rightarrow, \iota)$ be an ATS, let $\Sigma = (S, q_0)$, for $q_0 \in Q$, be a rooted ATS. We now address a few questions concerning the relationship of asynchronous transition systems as *acceptors* of trace languages, and trace languages as *computations* of asynchronous transition systems.

It is clear from definition of trace semantics that each derivation of Σ is contained in some computation belonging to $T^{\infty}(\Sigma)$. In order to show that trace semantics is an adequate representation of the behaviour of an asynchronous transition system, we need to prove that each representant of a computation in $T^{\infty}(\Sigma)$ is a derivation of Σ, and all equivalent finite derivations lead to the same state.

Proposition 3.6.1.

Let $S = (Q, A, \rightarrow, \iota)$ be an ATS, let $\Sigma = (S, q_0)$, for $q_0 \in Q$, be a rooted ATS.

(i) $\forall\, \sigma \in T^{\infty}(\Sigma)\colon\ \sigma \subseteq D^{\infty}(\Sigma)$

(ii) $\forall\, x, y \in D^{*}(\Sigma)\colon\ x \equiv_{\iota} y \Rightarrow ((q_0 \rightarrow^{x} q' \wedge q_0 \rightarrow^{y} q'') \Rightarrow (q' = q''))$

Proof.

(i) Suppose $\sigma \in T^{\infty}(\Sigma)$; then there exists $x \in \sigma$ such that $x \in D^{\infty}(\Sigma)$, and thus $q_0 \rightarrow^x$. Hence, there exists a sequence of states $q_0, q_1, \ldots q_i \ldots$ such that:

$$q_0 \rightarrow^{x(1)} q_1 \rightarrow^{x(2)} \ldots q_{i-1} \rightarrow^{x(i)} \ldots.$$

We need to show for any $y \in \sigma$ that there exists a sequence of states $q_0, q'_1, \ldots q'_i \ldots$ such that:

$$q_0 \rightarrow^{y(1)} q'_1 \rightarrow^{y(2)} \ldots q'_{i-1} \rightarrow^{y(i)} \ldots.$$

First, consider finite computations. Suppose $x = a$ for some $a \in A$ such that $[x] \in T^*(\Sigma)$. Thus $[x] = \{a\}$ and there exist states q_0, q_1 such that $q_0 \rightarrow^a q_1$. Let σ be of length $n > 1$ such that $\sigma \in T^*(\Sigma)$, then there exists $x \in \sigma$ such that $q_0 \rightarrow^x$ and x is of length n. Let $y \in \sigma$. Since $x \equiv y$, there exists a sequence of strings w_j, $1 \leq j \leq N$, such that

$$w_j = uabv$$

$$w_{j+1} = ubav$$

with $w_j = x$, $w_N = y$, $a \iota b$ and each $w_j \in \sigma$. Since $q_0 \rightarrow^x$, there exists a sequence of states $q_0 \rightarrow^{x(1)} q_1 \rightarrow^{x(2)} \ldots q_n$. We show that if $w_j \in D^*(\Sigma)$ then $w_{j+1} \in D^*(\Sigma)$, for any j, $1 \leq j \leq N$. Suppose $w_j \in D^{\infty}(\Sigma)$, then there exists a sequence of states

$$q_0 \rightarrow^{u(1)} q_1 \ldots q_i \rightarrow^a q_{i+1} \rightarrow^b q_{i+2} \rightarrow^{v(1)} \ldots q_n$$

Since $a \iota b$, it follows from condition (i) of the definition of an asynchronous transition system that there exists a state q' such that $q_i \rightarrow^b q' \rightarrow^a q_{i+2}$. Clearly, $w_{j+1} \in D^*(\Sigma)$. We did not require forward stability in the above proof.

Now consider infinite computations. Suppose $\sigma \in T^{\omega}(\Sigma)$, then there exists $u \in \sigma$ such that $u \in D^{\omega}(\Sigma)$, and thus $q_0 \rightarrow^u$. Hence, there exists an infinite sequence of states $q_0, q_1, \ldots q_i \ldots$ such that $q_0 \rightarrow^{u(1)} q_1 \rightarrow^{u(2)} \ldots q_{i-1} \rightarrow^{u(i)} \ldots$. Let $v \in \sigma$; then $v \leq^{\infty} u$ and, by definition of $\leq^{\infty}$, for every prefix x of v there exists a prefix y of u such that $x \leq^* y$. Let x be any finite prefix of v; then there exists $z \geq x$ such that $z \equiv_{\iota} y$. Since $y \in D^*(\Sigma)$ and $z \equiv_{\iota} y$, we have $z \in D^*(\Sigma)$. Thus we conclude that $v \in D^{\omega}(\Sigma)$ since each finite prefix x of v can be extended to a finite prefix z of v such that $z \in D^*(\Sigma)$.

(ii) Suppose $x, y \in D^*(\Sigma)$ such that $x \equiv y$. We need to show that $(q_0 \rightarrow^x q' \wedge q_0 \rightarrow^y q'')$ implies $q' = q''$. Suppose $x = a$ for some $a \in A$ such that $x \in D^*(\Sigma)$. Thus $[x] = \{a\}$ and there exist states q_0, q_1 such that $q_0 \rightarrow^a q_1$ (uniqueness of q_1 follows from non-ambiguity). Let $x, y \in D^*(\Sigma)$ be of length $n > 1$ such that $q_0 \rightarrow^x q' \wedge q_0 \rightarrow^y q''$. Since $x \equiv y$, there exists a sequence of strings w_j, $1 \leq j \leq N$, such that

$w_j = uabv$

$w_{j+1} = ubav$

with $w_j = x$, $w_N = y$, $a \iota b$ and each $w_j \in \sigma$. By part (i) of this proof we have $q_0 \rightarrow^{w_j}$. We show that if $q_0 \rightarrow^{w_j} q'$ and $q_0 \rightarrow^{w_{j+1}} q''$, for $1 \leq j \leq N$, then $q' = q''$. Suppose $q_0 \rightarrow^{w_j} q'$ and $q_0 \rightarrow^{w_{j+1}} q''$, then:

$q_0 \rightarrow^{u(1)} q_1 \ldots q_i \rightarrow^{a} q_{i+1} \rightarrow^{b} q_{i+2} \rightarrow^{v(1)} \ldots q'$

$q_0 \rightarrow^{u(1)} q_1 \ldots q_i \rightarrow^{b} q'_{i+1} \rightarrow^{a} q_{i+2} \rightarrow^{v(1)} \ldots q''$

Clearly, it follows from non-ambiguity that $q' = q''$, which concludes the proof.

❑

As a consequence of the above proposition we may now extend the notation to allow for a trace $\sigma \in \Theta_\iota^\infty$ to be *applicable* in a given state q, denoted $q \rightarrow^\sigma$, iff there exists $x \in D^\infty(\Sigma)$ such that $q \rightarrow^x$. Note that if x', x'' are finite prefixes of some x and $q \rightarrow^x$, then x' and x'' are comparable with respect to string prefix ordering. On the other hand, if σ', σ'' are finite prefixes of some σ and $q \rightarrow^\sigma$, then σ' and σ'' need *not* be comparable with respect to trace prefix ordering

The following proposition summarizes properties of the set of computations of a rooted asynchronous transition system. The set of admissible computations of a given system, $Max(T^\infty(\Sigma)) \cup T^*(\Sigma)$, has been adequately defined to characterize its every possible finite computation.

Proposition 3.6.2.

Let $S = (Q, A, \rightarrow, \iota)$ be an ATS, let $\Sigma = (S, q_0)$, for $q_0 \in Q$, be a rooted ATS.

(i) $T^\infty(\Sigma)$ is: prefix closed infinitary trace language, closed infinitary trace language and an ideal in Θ_ι^∞.

(ii) $Max(T^\infty(\Sigma)) \cup T^*(\Sigma)$ is an ideal in Θ_ι^∞.

Proof.

(i) We show $T^\infty(\Sigma)$ is prefix closed. Suppose $\sigma \in T^\infty(\Sigma)$ and let $\gamma \leq_\iota \sigma$. Choose $u \in \gamma$, then there exists $v \in \sigma$ such that $u \leq^\infty v$. Let $x \in Pref^{fin}(u)$. By definition of $\leq^\infty$:

$\exists\, y \in Pref^{fin}(v)$: $(\exists\, z: x \leq z \wedge z \equiv_\iota y) \Rightarrow$

$\exists\, y \in Pref^{fin}(v)$: $(\exists\, z: x \leq z \wedge z \equiv_\iota y \wedge q_0 \rightarrow^y)$ because $D^\infty(\Sigma)$ is prefix closed by Proposition 3.4.1(i). $\Rightarrow$

$\exists\, y \in Pref^{fin}(v)$: $(\exists\, z: x \leq z \wedge z \equiv_\iota y \wedge q_0 \rightarrow^z)$ by Proposition 3.6.1(i) $\Rightarrow$

$q_0 \rightarrow^x$ because $D^\infty(\Sigma)$ is prefix closed by Proposition 3.4.1.

Thus, we have shown $q_0 \rightarrow^x$ for any finite prefix x of u; since $D^\infty(\Sigma)$ is closed by Proposition 3.4.1 we conclude $q_0 \rightarrow^u$.
Finally, $[u] = \gamma \in T^\infty(\Sigma)$, and thus $T^\infty(\Sigma)$ is prefix closed.

Since every prefix closed trace language is an ideal, it follows that $T^\infty(\Sigma)$ is an ideal.

We show $T^\infty(\Sigma)$ is closed. Suppose $\sigma \in Adh(T^\infty(\Sigma))$, then:

$$Pref^{fin}(\sigma) \subseteq \Theta_\iota{}^* \cap T^\infty(\Sigma) = T^*(\Sigma).$$

Let $u \in \sigma$, x be an arbitrary finite prefix of u. Then $[x] \leq_\iota [u] = \sigma$. Hence, $q_0 \rightarrow^x$ by definition of $T^*(\Sigma)$. It follows that $q_0 \rightarrow^u$ because $D^\infty(\Sigma)$ is closed (Proposition 3.4.1) and $[u] = \sigma \in T^\infty(\Sigma)$.

(ii) Let $\sigma \in T^\infty(\Sigma)$ such that $\sigma \notin Max(T^\infty(\Sigma))$; then there exists $\tau \in Max(T^\infty(\Sigma))$ such that $\sigma \leq_\iota \tau$. Thus, $Pref^{fin}(\sigma) \subseteq Pref^{fin}(\tau) \subseteq T^*(\Sigma)$, and $Pref^{fin}(Max(T^\infty(\Sigma))\text{-}\{\sigma\}) = Pref^{fin}(Max(T^\infty(\Sigma)))$. It is easy to see $Pref^{fin}(Max(T^\infty(\Sigma))) = T^*(\Sigma)$.

❑

We can also characterize the set of computations of certain subclasses of asynchronous transition systems. Note that determinate systems allow concurrency, but disallow non-determinism in the sense of two dependent actions being enabled in the same state. On the other hand, sequential systems do not allow concurrency, but admit non-determinism.

Proposition 3.6.3.
Let $S = (Q, A, \rightarrow, \iota)$ be an ATS, let $\Sigma = (S, q_0)$, for $q_0 \in Q$, be a rooted ATS.
(i) If Σ is a determinate forward stable system, then $T^\infty(\Sigma)$ is a complete lattice.
(ii) If Σ is a sequential system, then any two consistent computations in $T^\infty(\Sigma)$ are comparable.

Proof.
(i) Let $\sigma, \gamma \in T^*(\Sigma)$ and let $\delta \in T^*(\Sigma)$ denote maximal common prefix of σ, γ. Observe that $\sigma/\delta, \gamma/\delta \in T^*(S)$. We show that $(\sigma/\delta)\ \iota\ (\gamma/\delta)$. If $\sigma/\delta = [\varepsilon]$ or $\gamma/\delta = [\varepsilon]$ then it is clear that $(\sigma/\delta)\ \iota\ (\gamma/\delta)$. Suppose $\sigma/\delta \neq [\varepsilon]$ and $\gamma/\delta \neq [\varepsilon]$, then there exist $[a] \leq \sigma/\delta$, $[b] \leq \gamma/\delta$ such that:
(a) $q \rightarrow^{[a]} q'$, $q \rightarrow^{[b]} q''$ and $q' \neq q''$ (because δ is a maximal common prefix)
where $q_0 \rightarrow^\delta q$. Since Σ is determinate, (a) implies $b\ \iota\ a$. By condition (ii) of the

definition of ATS (forward stability), there exists q''' such that:

$$q \rightarrow[a]\ q' \rightarrow[b]\ q''',\ q \rightarrow[b]\ q'' \rightarrow[a]\ q'''.$$

By induction on length of σ/δ we have $b\ \iota\ a$ for all $a \in Act(\sigma/\delta)$. Since $\delta[b](\sigma/\delta) \in T^*(\Sigma)$, the above may be repeated for $\gamma/(\delta[b])$. By induction on the length of γ/δ, it may be concluded that $Act(\sigma/\delta) \times Act(\gamma/\delta) \subseteq \iota$, and hence $(\sigma/\delta)\ \iota\ (\gamma/\delta)$.

It is easy to see that the least upper bound of σ and γ, for $\sigma, \gamma \in T^*(\Sigma)$ is $\delta(\sigma/\delta)(\gamma/\delta) \in T^*(\Sigma)$, where $\delta \in T^*(\Sigma)$ is the maximal common prefix of σ and γ. Thus, $T^*(\Sigma)$ contains least upper bounds of every pair of its elements, and is therefore directed. Hence, the least upper bound of every subset of $T^*(\Sigma)$ exists and belongs to $T^\infty(\Sigma)$ (because $T^\infty(\Sigma)$ closed by Proposition 3.6.2). Now, greatest lower bounds exist because $glb(X) = lub\{\sigma \in T^\infty(\Sigma) \mid \sigma \leq X\}$.

(ii) Follows from properties of string prefix ordering.

❑

Example. The ATS shown in Fig. 3.3.3 is determinate. Fig. 3.6.1 shows the complete lattice of computations of this system.

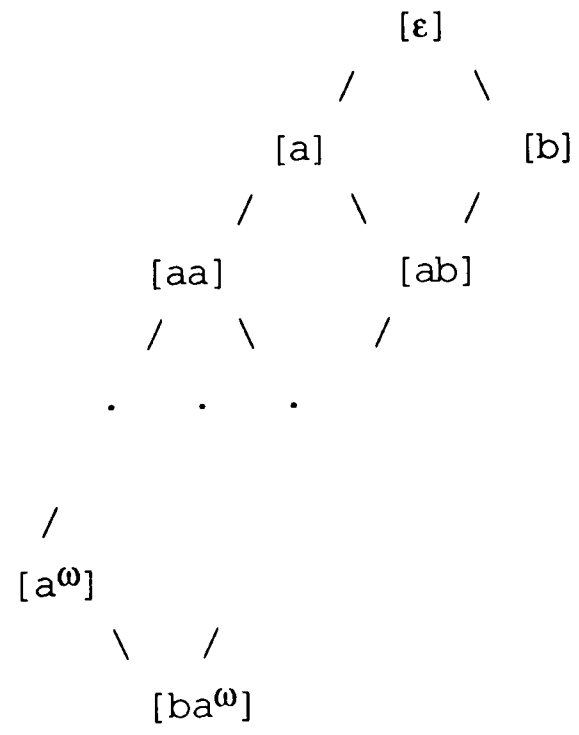


Fig. 3.6.1. The complete lattice of computations of the ATS shown in Fig. 3.3.3.

Example. The ATS shown in Fig. 3.3.6 is sequential. Fig. 3.6.2 shows the set of computations of this system.

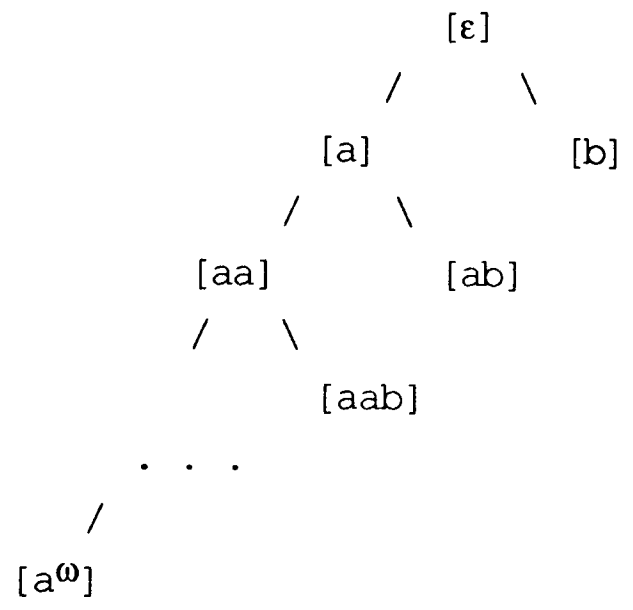


Fig. 3.6.2. The set of computations of the ATS shown in Fig. 3.3.6.

The following statement formally characterizes infinitary trace languages which are accepted by asynchronous transition systems. For any prefix-closed and closed infinitary trace language T there exists an asynchronous transition system that accepts it. However, in order to guarantee forward stability of the accepting ATS, we must, in addition, assume that that T contains least upper bounds of all its finite subsets that have a least upper bound. Note that when independency is empty, the proposition below specializes to the statement that for any prefix-closed and closed string language there exists a transition system accepting it.

Proposition 3.6.4.
Let $T \subseteq \Theta_\iota^\infty$ be a prefix-closed and closed infinitary trace language over a concurrent alphabet (A, ι). Then there exists an unambiguous rooted asynchronous transition system $\Sigma = (S, q_0)$ such that $T = T^\infty(\Sigma)$. If T contains least upper bounds of all finite subsets of T that have a least upper bound, then Σ is forward-stable.

Proof. The rooted ATS $\Sigma = (S, q_0)$, where $S = (Q, A, \rightarrow, \iota)$ can be constructed as follows:

$$Q = T^{fin}$$
$$q_0 = [\,\varepsilon\,]$$
$$\tau' \rightarrow^a \tau'' \Leftrightarrow \exists\, a \in A:\ \tau'|a| = \tau''.$$

1) We show that Σ is an ATS. Obviously, Q is countable.
(1a) Let $\tau, \tau', \tau'' \in Q$, $a, b \in A$. Suppose $\tau \rightarrow^a \tau' \rightarrow^b \tau''$ and $a \iota b$. Then $\tau[a] = \tau'$ and $\tau'[b] = \tau''$. Define $\tau''' = \tau[b]$. Clearly, $\tau'' = \tau'[b] = \tau[a][b] = \tau[ab] = \tau[ba] = \tau[b][a] = \tau'''[a]$, and hence $\tau \rightarrow^b \tau''' \rightarrow^a \tau''$. $\tau''' \in Q$ because $\tau''' \leq \tau''$ and T is prefix-closed.
(1b) Let $\tau, \tau', \tau'' \in Q$, $a, b \in A$. Suppose $\tau \rightarrow^a \tau'$ and $\tau \rightarrow^b \tau''$ and $a \iota b$. Then $\tau[a] = \tau'$ and $\tau[b] = \tau''$. Define $\tau''' = \tau[a][b]$. Clearly, $\tau''' = \tau'[b] = \tau''[a]$, and hence $\tau' \rightarrow^b \tau'''$ and $\tau'' \rightarrow^a \tau'''$. Note that $\tau''' \in Q$ only if $lub\{\tau',\tau''\} = \tau''' \in T$, thus Σ is forward-stable only if this condition holds.
2) Clearly, Σ is unambiguous.
3) It is easy to see that $T^{fin} = T^*(\Sigma)$ and $T = T^\infty(\Sigma)$ because T closed.

❑

A prefix-closed and closed infinitary trace language may contain infinite behaviours that are not maximal. Since systems (programs) evolve in finite steps, such behaviours are not reachable. Thus, the behaviour of such a system would correspond to a trace language that is an ideal, rather than prefix-closed and closed. It is an easy consequence of the above proposition that given an ideal T we can construct a rooted asynchronous transition system Σ such that:

$T^{fin} = T^*(\Sigma)$, $T \cup Adh(T) = T^\infty(\Sigma)$, $T^{fin} \cup Max(T^{inf}) \subseteq Adm(\Sigma)$.

Unfortunately, we cannot show the existence of an ATS Σ such that $T = T^\infty(\Sigma)$. Consider $T = [(a^*b^*)^*] \cup [(a^*bb^*)^\omega]$, which is an ideal but it is not closed. Then $Adh(T) = [(a^*b^*)^\omega] = T^\omega(\Sigma)$, but $T^\infty(\Sigma) \not\subset T$.

4

Defining Fairness for Non-Interleaving Concurrency

A *fairness notion* is introduced in the semantic models as a *constraint*, the purpose of which is to exclude those behaviours that do not impose the progress of certain components of the system, or as a *liveness property* that can be added to the list of properties that constitute the specification of the system. Fairness notions are relative to the granularity level and the strength required.

The purpose of this chapter is to investigate ways in which existing fairness notions could be translated into asynchronous transition systems and what benefits could be gained when employing trace semantics as opposed to the interleaving semantics. Our main concern is to define fairness with respect to concurrency, rather than non-determinism. In other words, we shall be concerned mainly with imposing fairness of conflicts that arise due to synchronisation and, as such, may result in some process being delayed while waiting to synchronise. Since asynchronous transition systems have been equipped with process structure, we have chosen to distinguish the granularity level of *events* and the level of *processes*. The strength predicates considered will be *weak*, *strong* and *unconditional* fairness.

This chapter will focus on pragmatic definitions of event and process fairness. An attempt to mathematically formalize fairness will be included in the following chapter.

4.1. How Not To Define Fairness

It might seem reasonable to directly translate existing fairness notions (see Chapter 2 for review) into asynchronous transition systems. These definitions have been formulated for interleaving semantics and, as a result, are based on single execution sequences rather than equivalence classes of execution sequences that were developed in Chapter 3. Execution sequences constitute *sequential observations* rather than *computations* and, as such, they correspond to derivations in our model.

Most commonly known definitions of event fairness were originally introduced in the context of interleaving semantics [LPS81] [Par81] and can be summarized as follows:

(i) unconditional fairness: *every event is taken infinitely often*;

(ii) strong fairness: *every event that becomes possible infinitely often is taken infinitely often*;

(iii) weak fairness: *every event that becomes possible from some point on is eventually taken.*

In order to obtain process fairness, the word *event* should be substituted with the word *process* in the above.

Fairness in the Sequential Sense

Although the above definitions do not state precisely what it means for an event to become possible, existing literature suggests the following straightforward approach. Let $\Sigma = (S, q_0)$ be a rooted ATS, where $S = (Q, A, \rightarrow, \iota)$ and $q_0 \in Q$. We assume transition labels $a \in A$ determine events within the system. Let $a \in A$, $x \in D^{\infty}(\Sigma)$ such that:

$$q_0 \rightarrow^{x(1)} q_1 \rightarrow^{x(2)} q_2 \ldots q_i \rightarrow^{x(i)} q_{i+1} \cdots$$

An event $a \in A$ *becoming possible* is defined as a transition labelled with a being enabled in a given state q ($q \rightarrow^a$). An event $a \in A$ being *taken* in a derivation x is defined as an occurrence of the symbol a in x; in other words, an event a is taken iff $\exists\, i\colon\ q_i \rightarrow^a q_{i+1}$.

Formally, the above definitions may now be given as follows (i_a denotes some natural number depending on a).

Definition. Let $S = (Q, A, \rightarrow, \iota)$ be an ATS, $\Sigma = (S, q_0)$ be a rooted ATS, and $x \in D^\omega(\Sigma)$ such that:

$$q_0 \rightarrow^{x(1)} q_1 \rightarrow^{x(2)} q_2 \ldots q_i \rightarrow^{x(i)} q_{i+1} \ldots$$

where $q_i \in Q$ for all i.

A derivation x is *unconditionally event fair in the sequential sense* (SQ-UEF) iff

$$\forall\, a \in A\ \forall\, k\ \ \exists\, i_a \geq k\colon\ q_{i_a} \rightarrow^a q_{i_a+1}.$$

A derivation x is *strongly event fair in the sequential sense* (SQ-SEF) iff

$$\forall\, a \in A\ \forall\, k\colon\ (\forall\, m > k\ \exists\, j \geq k\colon\ q_j \rightarrow^a) \Rightarrow (\exists\, i_a \geq k\colon q_{i_a} \rightarrow^a q_{i_a+1}).$$

A derivation x is *weakly event fair in the sequential sense* (SQ-WEF) iff

$$\forall\, a \in A\ \forall\, k\colon\ (\forall\, m > k\colon\ q_m \rightarrow^a) \Rightarrow (\exists\, i_a \geq k\colon q_{i_a} \rightarrow^a q_{i_a+1})$$

Any *finite* derivation ($x \in D^*(\Sigma)$) is *fair in the sequential sense*. Note that unconditional fairness should only be used when transitions may not become disabled in the course of computation. Strong event fairness requires that a transition be taken infinitely often if a state which enables this transition is reached infinitely often in a given derivation. Weak event fairness disallows a derivation if some transition is enabled in every state visited from some point on and never taken.

Example. Let us consider the ATS shown in Fig. 3.3.3. The derivation ba^ω is SQ-WEF and SQ-SEF but not SQ-UEF. The derivation a^ω is unfair according to each of the above definitions. Similarly, the derivation a^ω of the ATS pictured in Fig. 3.3.6 is unfair with respect to each of the definitions presented here.

It can be shown that the above definitions form a hierarchy. Let us define:

$$\text{SQ-UEFair}(\Sigma) = \{ x \in D^{\infty}(\Sigma) \mid x \text{ is SQ-UEF} \}$$
$$\text{SQ-SEFair}(\Sigma) = \{ x \in D^{\infty}(\Sigma) \mid x \text{ is SQ-SEF} \}$$
$$\text{SQ-WEFair}(\Sigma) = \{ x \in D^{\infty}(\Sigma) \mid x \text{ is SQ-WEF} \}$$

Proposition 4.1.1. (Hierarchy of Fairness in the Sequential Sense)
$\text{SQ-UEFair}(\Sigma) \subset \text{SQ-SEFair}(\Sigma) \subset \text{SQ-WEFair}(\Sigma)$, and the inclusion is strict.

Proof. SQ-UEF $\Rightarrow$ SQ-SEF $\Rightarrow$ SQ-WEF is clear.
The following serves as a counter-example to SQ-WEF $\Rightarrow$ SQ-SEF. Let us consider the ATS shown in Fig. 4.1.1 where $\iota = \varnothing$. The infinite derivation:

$$q' \rightarrow^{a} q'' \rightarrow^{c} q' \rightarrow^{a} q'' \rightarrow^{c} \ldots$$

is *not* SQ-SEF with respect to b (because b is enabled infinitely often and never taken), but it is SQ-WEF (because b is not enabled continuously).

A counter-example to SQ-SEF $\Rightarrow$ SQ-UEF is shown in Fig. 4.1.2, where $\iota = \varnothing$. The infinite derivation $q'' \rightarrow^{a} q' \rightarrow^{c} q' \rightarrow^{c} q' \rightarrow^{c} \ldots$ is *not* SQ-UEF with respect to b (because b is not taken infinitely often), but it is SQ-SEF (because b is not enabled infinitely often). This concludes the proof.

❑

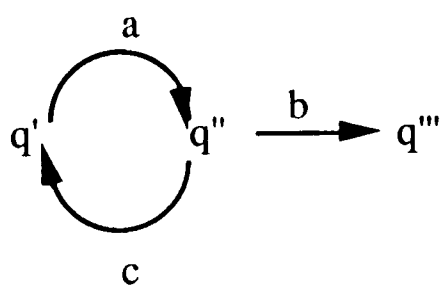


Fig. 4.1.1. A counter-example to SQ-WEF $\Rightarrow$ SQ-SEF ($\iota = \varnothing$).

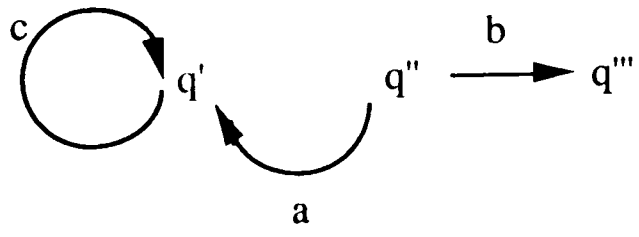


Fig. 4.1.2. A counter-example to SQ-SEF $\Rightarrow$ SQ-UEF ($\iota = \varnothing$).

Relationship to Trace Semantics

However, the seemingly straightforward adaptation of the above definitions causes serious problems: we observe that SQ-WEF is not closed under trace equivalence (not *equivalence robust* using the terminology of [AFK87]). In other words, it is possible to show two equivalent derivations (with respect to the trace equivalence $\equiv_t$), one of which will be fair and the other one unfair. Let us consider the following counter-example for SQ-SEF (a similar one can be found for SQ-WEF) shown in Fig. 4.1.3, where a ι d, c ι d, a ι e, c ι e.

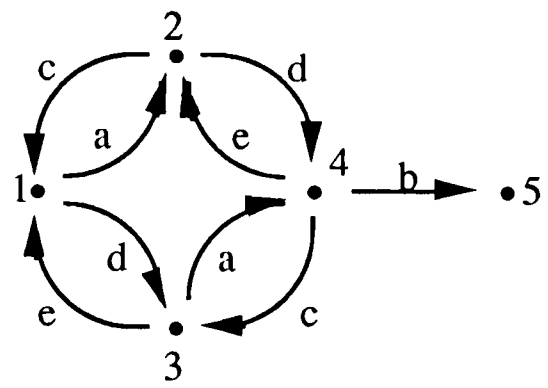


Fig. 4.1.3. An ATS that demonstrates that SQ-SEF is not closed under trace equivalence.

Then $(acde)^\omega \equiv_t (adce)^\omega$, and $(adce)^\omega$ is *not* strongly event fair in the sequential sense (SQ-SEF) because b is infinitely often enabled, but never taken. On the other hand, $(adce)^\omega$ *is* strongly event fair (SQ-SEF) because b is never enabled. It should be noted that the trace $[(acde)^\omega]$, which incorporates $(adce)^\omega$, is an admissible computation of the system pictured above, that is, it is maximal in $T^\infty(S)$; so are each of the finite computations $[(acde)^*adb]$. However, the sequence of states determined by the derivation $(acde)^\omega$ differs from the sequence of states determined by $(adce)^\omega$, and, as a consequence, statements made about the enabledness of a transition in a state visited in one derivation do not necessarily apply to equivalent derivations. When considering trace semantics, it can be easily verified that an infinite number of finite prefixes of the trace $[(adce)^\omega]$, namely $[(adce)^*ad]$, can be extended with b to form the trace $[(adce)^*adb]$ which is incomparable with $[(adce)^\omega]$. (When attempting to extend $[(adce)^\omega]$ with b the resulting trace is $[(adce)^\omega]$).

Relationship to Confusion

The above problem seems impossible to reconcile when behaviours of systems are represented as sets of sequences of transitions (i.e. interleaving semantics), so one might be tempted to restrict the model so that situations like the above could be disallowed. Unfortunately, this would greatly affect the generality of the model as the difficulty is caused by a phenomenon called *confusion* (see Chapter 3). Confusion arises in net theory as well as in asynchronous transition systems. In the world of hardware, it presents itself as the "glitch" problem [RoT86]. In fact, the system above has been derived from a Petri net that exhibits confusion. The original net (Condition/Event net) is shown in Fig. 4.1.4. It can be shown that (unlabelled) Condition/Event nets determine (unambiguous) asynchronous transition systems (see Chapter 6), where the set of all the states Q is defined as the set of all the cases, the set of action labels A is taken as the set of events of the net, the transition relation $\rightarrow$ is the firing relation, and the independency ι, determined by the structure of the net, defines the events that could occur autonomously. Thus, the firing sequences of a net exactly correspond to the derivations of asynchronous transition systems.

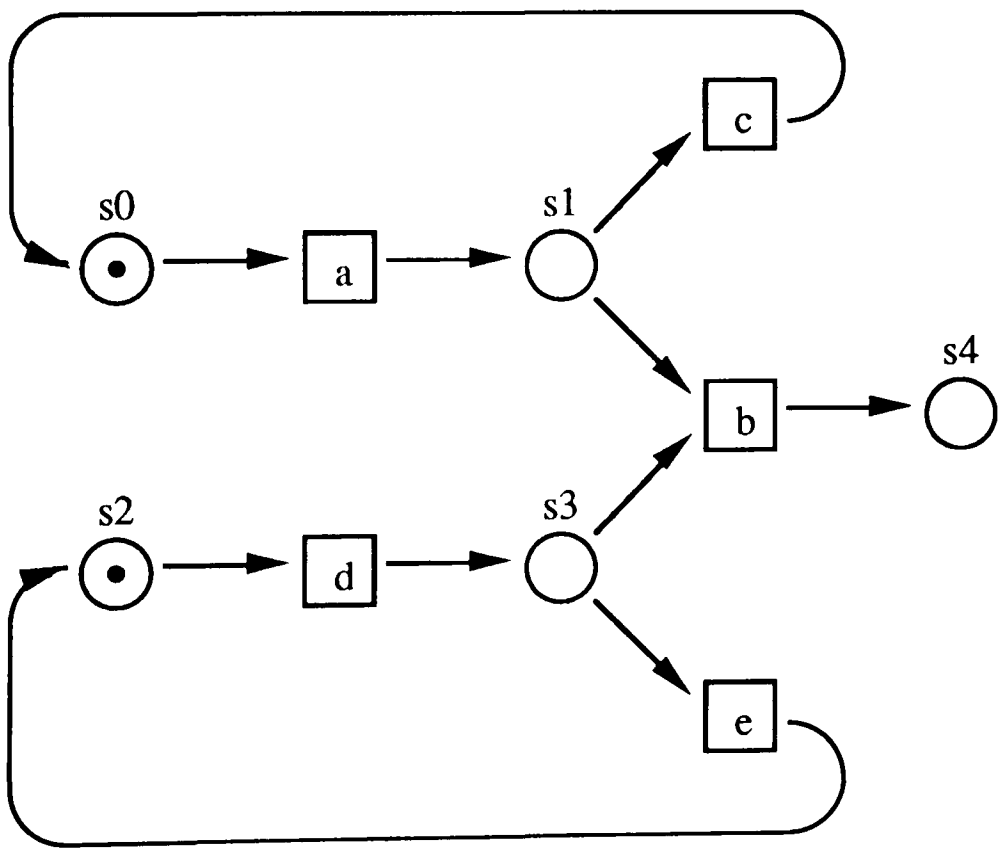


Fig. 4.1.4. A Condition/Event net that determines the ATS shown in Fig. 4.1.3.

The following are asymmetric confusions that the ATS shown in Fig. 4.1.3 exhibits:

$\alpha_1 = (2, c, d)$
$\alpha_2 = (3, e, a)$

because the conflict sets are as follows:

$cfl(2, c) = \varnothing$ $\qquad$ $cfl(3, e) = \varnothing$
$cfl(4, c) = \{b\}$ $\qquad$ $cfl(4, e) = \{b\}$

$cfl(2, d) = \varnothing$ $\qquad$ $cfl(3, a) = \varnothing$
$cfl(1, d) = \varnothing$ $\qquad$ $cfl(1, a) = \varnothing$

Confusion is a situation, in which conflict sets of a given transition (i.e. the set of transitions with which the given transition is at conflict) are *changed* through an occurrence of some transition concurrent to the one under consideration. Thus, different sequentializations of essentially the same behaviour may differ radically in terms of non-deterministic choices available at each step. It is not sufficient, in such situations, to base a definition on just one sequentialization, but it is necessary to consider *all* the possible sequentializations of a given behaviour; otherwise, a reasonable scheduler could not be built.

Although sequential systems, amongst which we can include CCS with standard transition system semantics [Mil80], are free from confusion, it is nevertheless possible to give CCS an interpretation in terms of asynchronous transition systems. For example, a CCS expression that determines the ATS shown in Fig. 4.1.3 may be $(p \parallel q)\backslash b$, where $p = fix\ X.(a(cX+bNIL))$, $q = fix\ X.(d(eX+\overline{b}))$. Thus asynchronous CCS also exhibits confusion.

It should be mentioned here that confusion, although it has been known in net theory for some time, can nevertheless cause many headaches. In [Bes84a] [Bes84b] an attempt has been made to adapt fairness definitions as introduced in [LPS81] to the Petri net environment - needless to say, enabledness of a transition was based on the existence of states enabling it in a derivation. The result was an infinite number of classes of fairness that collapsed to a single notion when a simpler class of nets was assumed. As it has been observed by Shields [Shi88b], this hierarchy of fairness is artificial because it is related to the existence of confusion.

Summary

It has been shown that the notion of a transition being infinitely often enabled in a derivation that is based on the *existence* of an infinite number of *states* enabling a is not adequate in the context of non-interleaving concurrency. The solution which makes it possible for fairness to be closed under trace equivalence requires an alternative notion of transitions being enabled: they should be based on a *computation*, that is an equivalence class of *all* the possible sequentializations of a behaviour, rather than on a single derivation.

4.2. Event Fairness

We shall now present the definitions of event fairness suitable for asynchronous transition systems. We take the view that *no possible event should be delayed indefinitely*. Again, we shall assume that events are represented by transitions.

Preliminary Definitions

Let $\Sigma = (S, q_0)$ be a rooted ATS, where $S = (Q, A, \rightarrow, \iota)$ and $q_0 \in Q$. Let $\sigma \in T^{\infty}(S)$, $a \in A$. Note that by Proposition 3.6.1 a computation $\sigma \in T^{*}(S)$, when applied in a state q, uniquely determines a state q' such that $q \rightarrow^{\sigma} q'$. We define an event $a \in A$ *becoming possible* as a transition being enabled by a given computation $\sigma \in T^{*}(S)$ in the sense that $\sigma[a]$ is also a computation of S (that is, computation σ can be extended with action a). An event $a \in A$ being *taken* is defined as an occurrence of the symbol a in a computation.

We now formalize the notion of an event being taken, enabled continuously and infinitely often. Although the definitions presented here refer to asynchronous transition system $S = (Q, A, \rightarrow, \iota)$, they also apply to the rooted ATS because $T^{\infty}(\Sigma) \subseteq T^{\infty}(S)$.

Definition.
Let $S = (Q, A, \rightarrow, \iota)$ be an ATS, $\sigma \in T^{\omega}(S, q)$, $a \in A$.

(i) We say that a is *taken* in σ iff

$$\exists\, \gamma \in \mathrm{Pref}^{\mathrm{fin}}(\sigma)\colon \gamma[a] \leq_\iota \sigma.$$

(ii) We say that a is *continuously enabled* in σ iff

$$\forall\, \gamma \in \mathrm{Pref}^{\mathrm{fin}}(\sigma)\colon \gamma[a] \in T^*(S).$$

(iii) We say that a is *infinitely often enabled* in σ iff

$$\mathrm{card}\,\{\gamma \in \mathrm{Pref}^{\mathrm{fin}}(\sigma) \mid \gamma[a] \in T^*(S)\} = \omega.$$

In other words, event a is taken in a computation if, and only if, some finite prefix of this computation can be found so that, when extended with a, the resulting computation is a finite prefix of the original computation. We say that a is continuously enabled in a computation if, and only if, every finite prefix of this computation can be extended with a. We say that a is infinitely often enabled in an infinite computation if, and only if, an infinite number of finite prefixes of that computation can be extended with a.

Hierarchy of Event Fairness

Let $\sigma \in T^\omega(S)$, $\tau \in T^*(S)$ such that $\tau \leq_\iota \sigma$. Thus there exist $q_1, q_2 \in Q$ such that $q_1 \rightarrow^\sigma$ and $q_1 \rightarrow^\tau q_2$. It now follows that $q_1 \rightarrow^\tau q_2 \rightarrow^{(\sigma/\tau)}$; hence, $\sigma/\tau \in T^\omega(S, q_2) \subseteq T^\omega(S)$. We can therefore apply the definitions introduced in the previous section with respect to σ/τ.

Definition.
Let $S = (Q, A, \rightarrow, \iota)$ be an ATS, $\Sigma = (S, q_0)$ be a rooted ATS, $\sigma \in T^\omega(\Sigma)$, $a \in A$.

(i) σ is *unconditionally event fair* (UEF) iff

$$\forall\, a \in A\ \forall\, \tau \in \mathrm{Pref}^{\mathrm{fin}}(\sigma)\colon$$
a is taken in σ/τ.

(ii) σ is *strongly event fair* (SEF) iff

$$\forall\, a \in A\ \forall\, \tau \in \mathrm{Pref}^{\mathrm{fin}}(\sigma)\colon$$
a infinitely often enabled in $\sigma/\tau \Rightarrow$ a taken in σ/τ.

(iii) σ is *weakly event fair* (WEF) iff

$$\forall\, a \in A\ \ \forall\, \tau \in \mathrm{Pref}^{fin}(\sigma):$$
$$\text{a continuously enabled in } \sigma/\tau \Rightarrow \text{a taken in } \sigma/\tau.$$

As usual, any *finite* computation ($\sigma \in T^*(\Sigma)$) is *fair* with respect to any of the above definitions.

Note that the definition of weak event fairness disallows a situation, in which every prefix of a computation is extendable with action a, but a is never taken. Strong event fairness, on the other hand, does not admit a computation which has an infinite number of prefixes extendable with a, but a is not taken after some finite prefix. Unconditional event fairness imposes a fairmerge of an infinite number of instances of each event, and it should be used only when events may not become disabled.

Example. The computation $[(adce)^\omega]$ of the ATS shown in Fig. 4.1.1 is not SEF because an infinite number of its prefixes, namely each of $[(adce)^*ad]$, can be extended with b (nor is it UEF). However, $[(adce)^\omega]$ is WEF because it not permanently extendable with b. The computation $[a^\omega]$ of the ATS shown in Fig. 3.3.3 is not WEF, nor is it SEF nor UEF, because its every finite prefix $[a^*]$ can be extended with b. A similar observation can be made with respect to the computation $[a^\omega]$ of the ATS in Fig.3.3.6 concerning extendability of $[a^*]$ with τ_b.

It can be shown that the above definitions form a hierarchy. Let us define:

$$\mathrm{UEFair}(\Sigma) = \{\sigma \in T^\infty(\Sigma) \mid \sigma \text{ is UEF}\}$$
$$\mathrm{SEFair}(\Sigma) = \{\sigma \in T^\infty(\Sigma) \mid \sigma \text{ is SEF}\}$$
$$\mathrm{WEFair}(\Sigma) = \{\sigma \in T^\infty(\Sigma) \mid \sigma \text{ is WEF}\}$$

Proposition 4.2.1. (Hierarchy of Event Fairness)
$\mathrm{UEFair}(\Sigma) \subset \mathrm{SEFair}(\Sigma) \subset \mathrm{WEFair}(\Sigma)$, and the inclusion is strict.

Proof. UEF $\Rightarrow$ SEF $\Rightarrow$ WEF is clear.
The counter-example to SEF $\Rightarrow$ WEF is shown in Fig. 4.1.1 ($\iota = \varnothing$). The computation $[(ac)^\omega]$ is not SEF with respect to b (because its prefixes $[(ac)^*a]$ can be extended with b), but it is WEF (because its prefixes $[a(ca)^*]$ are not extendable with b).

The counter-example to UEF $\Rightarrow$ SEF is shown in Fig. 4.1.2 ($\iota = \varnothing$). The computation

$[ac^{\omega}]$ is not UEF with respect to b (because b is not taken infinitely often), but it is SEF (because no prefix of this computation can be extended with b).

This concludes the proof. Counter-examples with non-empty independency can also be found.

□

Applicability of Event Fairness

However, the above definitions may be considered too restrictive, as they enforce fairness of choice of transitions whenever a group of transitions becomes sufficiently often enabled in the course of computation. In other words, in case some transition becomes infinitely often enabled simultaneously with another one, both transitions would have to be taken infinitely often in order to guarantee fairness with respect to SEF. (Note that the precise context under which transitions become enabled simultaneously, that is whether they are independent or dependent, is ignored by the definition of SEF). In practice, we may wish to treat conflicts that arise due to process synchronisation differently from conflicts that correspond to internal non-determinism.

Let us discuss the following example based on a Condition/Event net shown in Fig. 4.2.1 that implements mutual exclusion. Consider trace semantics of the net shown in Fig. 4.2.1. It is easy to see that all admissible computations of this net can be given by the expression $[((abc)^{*}(def)^{*})^{\omega}]$ and all their finite prefixes. It follows that computations $[(abc)^{\omega}]$ and $[(def)^{\omega}]$ are also admissible. The latter would be highly undesirable if the net is meant to represent a solution to the mutual exclusion of two processes with alphabets {a,b,c} and {d,e,f}, where a, d are events representing that entry to the critical section has been granted, b,e are critical sections, and c, f represent local sections of code of each process respectively. Thus, in the computation $[(abc)^{\omega}]$, the process identified by the alphabet {d,e,f} is never allowed entry. It should be pointed out that the two processes, although concurrent, are *not* independent, and this is why their progress is not guaranteed by the admissibility (i.e. maximality with respect to trace prefix ordering) of computations.

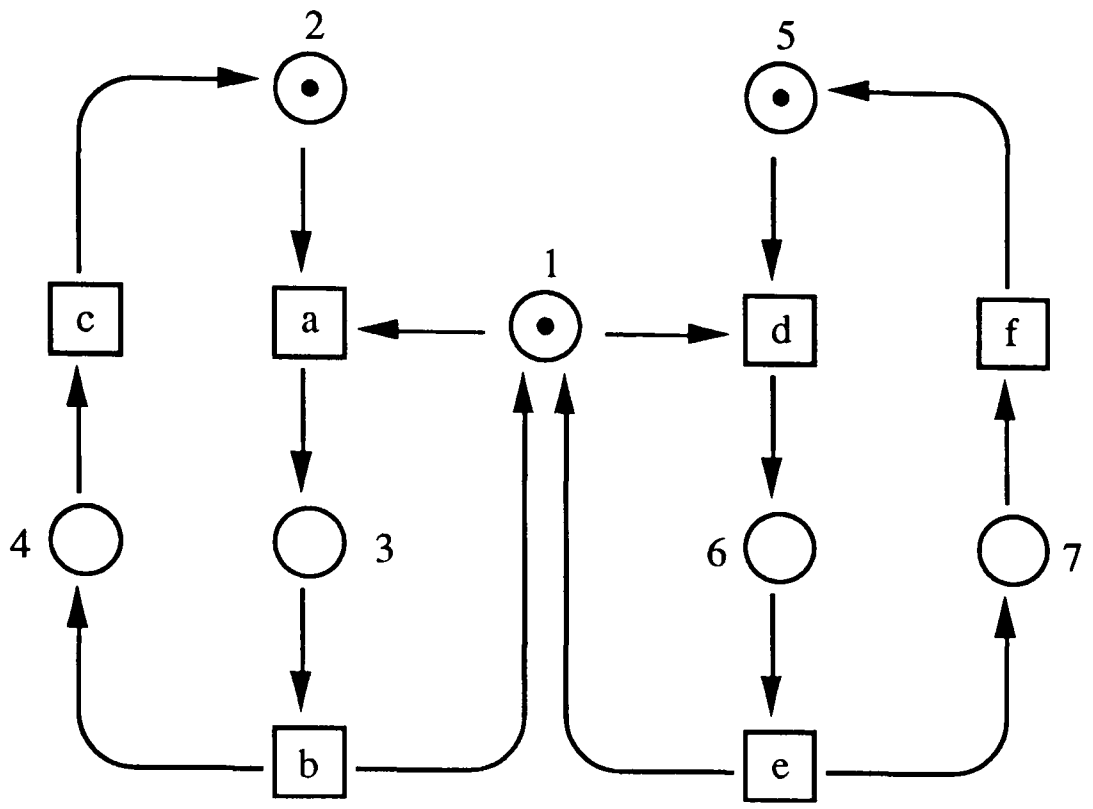


Fig. 4.2.1. A Condition/Event net that implements mutual exclusion, where a ι f, b ι f, c ι f, c ι d, c ι e.

It is an interesting question to determine whether confusion-free systems are free from the above difficulty. The net does, indeed, exhibit asymmetric confusions:

$\alpha_1 = (\{1,2,7\}, a, f)$
$\alpha_2 = (\{1,5,4\}, d, c)$

because a is at conflict with d after f has occurred, but not before (the same holds for d and c). However, confusion is *not* a major contributor to the difficulty under consideration. This fact could be demonstrated by removing transitions c and f; the resulting net is sequential and confusion-free, but the computations $[(abc)^\omega]$ and $[(def)^\omega]$ are still admissible. (Note that such problem does not arise in determinate systems.)

The above example shows that notions of fairness stronger than just the admissibility (maximality) of computations will be needed in practice. This could be used in favour of the definitions of event fairness we have introduced in this section. Note that the computation $[(abc)^\omega]$, although admissible and WEF, is *not* SEF because an infinite number of its finite prefixes $|(abc)^*|$ can be extended with d. In fact, the only computations of this net that are SEF must contain an infinite number of actions from each process.

Nevertheless, the above definitions of event fairness are still at fault: they may be considered too discriminating to be used generally. Let us consider the modification of the net from Fig. 4.2.1 which is shown in Fig. 4.2.2 (example due to A. Pnueli). In the modified net, there are two concurrent processes that compete for access to the critical section, but the process on the right may *internally* choose whether to request to enter it or execute its local section g. Since the choice is internal and non-deterministic, it is possible for the process on the right never to request entry to the critical section. Thus, the computation $[(abc(abc)^*gf(gf)^*)^\omega]$ is intuitively acceptable here. However, this computation, although admissible, is not SEF because an infinite number of its prefixes $[(abc(abc)^*gf(gf)^*)^*]$ can be extended with d.

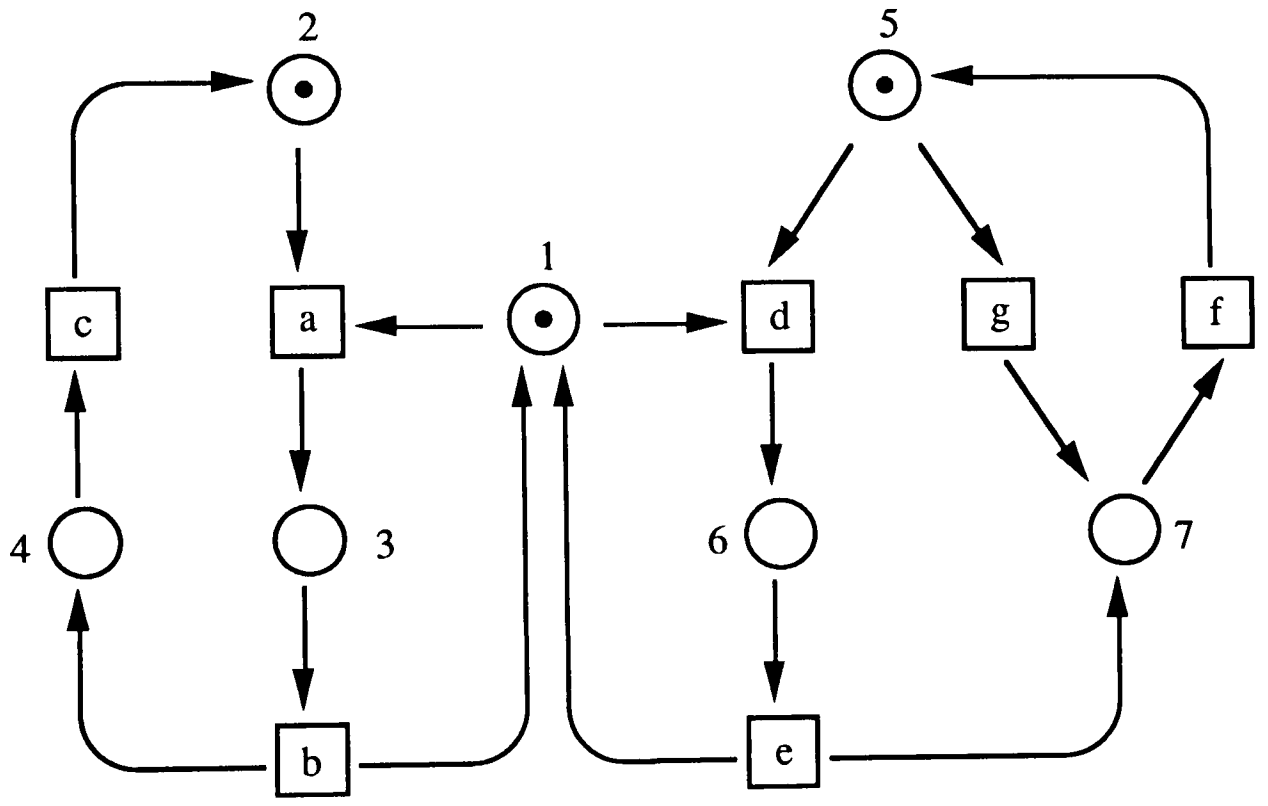


Fig. 4.2.2. A modified net that implements mutual exclusion, where a ι f, b ι f, c ι f, c ι d, c ι e, a ι g, b ι g, c ι g.

Summary

The main reason of the difficulty with the above notions of fairness is the fact that we attach the same weight to the conflicts arising due to the *synchronisation*, or *communication*, of concurrent processes as well as *internal choice*. (The dependency relation does not

distinguish between the two.) What would seem to provide a satisfactory solution is a way of making fairness notions relative to the conflicts that do require fair resolution; the remaining conflicts would not be affected by fairness at all. One way of doing this for asynchronous transition systems could be by introducing a more flexible notion of an event - after all, as stated in [Plo82], fairness depends entirely on what constitutes an event. It is possible to define a generalized notion of event fairness [Kwi88b] relative to *progress requirements*. The purpose of progress requirements is to explicitly describe those, and only those, conflicts that need to be fairly resolved. We have decided not to include this definition here as progress requirements roughly correspond to alphabet structures over a given concurrent alphabet.

4.3. Process Fairness

We shall now formalize definitions of process fairness suitable for asynchronous transition systems. We take the view that no process that becomes possible sufficiently often should be delayed indefinitely.

Preliminary Definitions

Let $\Sigma = (S, q_0)$ be a rooted ATS, where $S = (Q, A, \rightarrow, \iota)$ and $q_0 \in Q$. Let $\Pi = (\pi, \alpha)$ be a process structure over S. Let $\alpha = \{\alpha_1, \alpha_2, \ldots \alpha_n\}$. Let $\sigma \in T^{\infty}(S)$, and $\alpha_i \in \alpha$. We say a process α_i *becomes possible* after a computation $\sigma \in T^{*}(S)$ if there exists $a \in \alpha_i$ such that $\sigma[a]$ is also a computation of S (that is, computation σ can be extended with action a belonging to the process). A process α_i is *taken* if some action from its alphabet is taken.

We now formalize the notion of a process being taken, enabled continuously and infinitely often. These definitions also apply to rooted ATS.

Definition.
Let $S = (Q, A, \rightarrow, \iota)$ be an ATS and let $\Pi = (\pi, \alpha)$, with $\alpha = \{\alpha_1, \alpha_2, \ldots \alpha_n\}$, be a process structure over S. Let $\sigma \in T^{\omega}(S, q)$ and $\alpha_i \in \alpha$.

(i) We say that process α_i is *taken* in σ iff

$$\exists\, a \in \alpha_i,\ \exists\, \gamma \in \mathrm{Pref}^{fin}(\sigma)\colon \gamma[a] \leq_\iota \sigma.$$

(ii) We say that process α_i is *continuously enabled* in σ iff

$$\forall\, \gamma \in \mathrm{Pref}^{fin}(\sigma),\ \exists\, a \in \alpha_i\colon \gamma[a] \in T^*(S).$$

(iii) We say that process α_i is *infinitely often enabled* in σ iff

$$\mathrm{card}\ \{\gamma \in \mathrm{Pref}^{fin}(\sigma) \mid \exists\, a \in \alpha_i\colon\ \gamma[a] \in T^*(S)\} = \omega.$$

In other words, process α_i is taken in a computation if, and only if, some some action $a \in \alpha_i$ is taken. Alternatively, we could use process projections to define the notion of the process α_i being taken in σ, for example as:

$$| p_i^{\alpha}(\sigma) | \neq 0.$$

We say that process α_i is continuously enabled in a computation if, and only if, every finite prefix of this computation can be extended with some $a \in \alpha_i$. We say that process α_i is infinitely often enabled in a computation if, and only if, an infinite number of finite prefixes of that computation can be extended with some $a \in \alpha_i$.

It should be noted that since there is a variety of differing alphabet structures available, and process structures are determined by alphabet structures, the above notions are relative to the process structure chosen. For example, the ATS shown in Fig. 3.3.3, with $a\ \iota\ b$, allows only one process structure determined by the alphabet structure $\{\{a\},\{b\}\}$. Thus, process $\{b\}$ is continuously enabled in $[a^\omega]$ but never taken. On the other hand, the ATS pictured in Fig. 3.3.6, where a is dependent on τ_b, allows four process structures, for example $\{\{a,\tau_b\}\}$ and $\{\{a,\tau_b\},\{\tau_b\}\}$. Observe that both processes $\{a,\tau_b\}$ and $\{\tau_b\}$ are continuously enabled in a computation $[a^\omega]$, but only process $\{\tau_b\}$ is never taken.

Also note that if $\alpha_i \subseteq \alpha_j$ for some alphabets over (A, ι), then α_i enabled (taken) in a given computation implies α_j enabled (taken).

Strength Hierarchy of Process Fairness

Definition.
Let $S = (Q, A, \rightarrow, \iota)$ be an ATS, $\Sigma = (S, q_0)$ be a rooted ATS, and let $\Pi = (\pi, \alpha)$, with $\alpha = \{\alpha_1, \alpha_2, \ldots \alpha_n\}$, be a process structure over S. Let $\sigma \in T^\infty(\Sigma)$ and $\alpha_i \in \alpha$.

(i) σ is *unconditionally process fair wrt* Π, denoted UPF(Π), iff

$\forall\, \alpha_i \in \alpha\ \forall\, \tau \in \mathrm{Pref}^{fin}(\sigma)$:
α_i taken in σ/τ.

(ii) σ is *strongly process fair wrt* Π, denoted SPF(Π), iff

$\forall\, \alpha_i \in \alpha\ \forall\, \tau \in \mathrm{Pref}^{fin}(\sigma)$:
α_i infinitely often enabled in $\sigma/\tau \Rightarrow \alpha_i$ taken in σ/τ.

(iii) σ is *weakly process fair wrt* Π, denoted WPF(Π), iff

$\forall\, \alpha_i \in \alpha\ \forall\, \tau \in \mathrm{Pref}^{fin}(\sigma)$:
α_i continuously enabled in $\sigma/\tau \Rightarrow \alpha_i$ taken in σ/τ.

As usual, any *finite* computation ($\sigma \in T^*(\Sigma)$) is *fair* with respect to any of the above definitions.

The above definitions are similar to the definitions of event fairness. Weak process fairness disallows a computation in which some process is continuously enabled but never taken. Strong process fairness relaxes the premise to some process being enabled infinitely often. Unconditional process fairness imposes a fairmerge of actions of all processes.

It can be shown that for a given process structure the above definitions form a hierarchy. Let us define:

$$\mathrm{UPFair}(\Sigma,\Pi) = \{\sigma \in T^\infty(\Sigma) \mid \sigma \text{ is UPF wrt } \Pi\}$$
$$\mathrm{SPFair}(\Sigma,\Pi) = \{\sigma \in T^\infty(\Sigma) \mid \sigma \text{ is SPF wrt } \Pi\}$$
$$\mathrm{WPFair}(\Sigma,\Pi) = \{\sigma \in T^\infty(\Sigma) \mid \sigma \text{ is WPF wrt } \Pi\}$$

Proposition 4.3.1. (Strength Hierarchy of Process Fairness)
$\mathrm{UPFair}(\Sigma,\Pi) \subset \mathrm{SPFair}(\Sigma,\Pi) \subset \mathrm{WPFair}(\Sigma,\Pi)$, and the inclusion is strict.

Proof. UPF(Π) $\Rightarrow$ SPF(Π) $\Rightarrow$ WPF(Π) is clear.
The counter-example to SPF(Π) $\Rightarrow$ WPF(Π) is shown in Fig. 4.1.1 ($\iota = \varnothing$). Let α = {{a,b,c}, {b,c}, {a,b}, {b}}. The computation $[(ac)^\omega]$ is not SPF(Π) with respect to process {b} (because its prefixes $[(ac)^*a]$ can be extended with b), but it is WPF(Π) (because its prefixes $[a(ca)^*]$ are not extendable with b).

The counter-example to UPF(Π) $\Rightarrow$ SPF(Π) is shown in Fig. 4.1.2 ($\iota = \varnothing$). Let α = {{a,b,c}, {b,c}, {a,b}, {b}}. The computation [ac$^\omega$] is not UPF(Π) with respect to {b} (because b is not taken infinitely often), but it is SPF(Π) (because no prefix of this computation can be extended with b).

This concludes the proof. Counter-examples with non-empty independency can also be found.

❑

Example. Let us consider trace semantics of the Condition/Event net shown in Fig. 4.2.1. The minimal alphabet structure here is:

$$\alpha^{min} = \{\{a,b,d,e\},\{a,b,c\},\{d,e,f\}\}$$

Let $\Pi^{min} = (\pi, \alpha^{min})$. The (admissible) computation [(abc)$^\omega$] is *not* SPF(Π^{min}) with respect to process {d,e,f} because each prefix [(abc)*] can be extended with d, but it is WPF(Π^{min}) as the process {d,e,f} is *not* enabled continuously. Likewise, all computations of the form [((abc)*(def)*)$^\omega$] where either {a,b,c} or {d,e,f} is taken finitely often are WPF(Π^{min}), but not SPF(Π^{min}). Computations [(abc(abc)*def(def)*)$^\omega$] are SPF(Π^{min}) and WPF(Π^{min}). In fact, they also are UPF(Π^{min}).

Note that since all finite computations are fair, the computations [(abc)*] and [(def)*] are UPF(Π^{min}), SPF(Π^{min}) and WPF(Π^{min}).

Refinement Hierarchy of Process Fairness

The above hierarchy of process fairness of different strength has been formulated for notions of fairness determined by the same process (or alphabet) structure. The hierarchy of process fairness can be extended onto fairness notions determined by process structures which are related through the refinement relation. Let S = (Q, A, $\rightarrow$, ι) be an ATS, Σ = (S, q_0) be a rooted ATS, and let $\Pi = (\pi, \alpha)$, $\Pi' = (\pi', \alpha')$ be process structures over S. Let $\alpha = \{\alpha_1, \alpha_2, \ldots \alpha_n\}$, $\alpha' = \{\alpha_1', \alpha_2', \ldots \alpha_k'\}$. The following can then be shown.

Proposition 4.3.2. (Refinement Hierarchy of Process Fairness)
If Π' is a refinement of Π (i.e. $\Pi \leq \Pi'$) then the following holds:

(i) UPF(Π') $\Rightarrow$ UPF(Π), i.e. UPFair(Σ,Π') $\subseteq$ UPFair(Σ,Π)

(ii) $SPF(\Pi') \Rightarrow SPF(\Pi)$, i.e. $SPFair(\Sigma,\Pi') \subseteq SPFair(\Sigma,\Pi)$

(iii) $WPF(\Pi') \Rightarrow WPF(\Pi)$, i.e. $WPFair(\Sigma,\Pi') \subseteq WPFair(\Sigma,\Pi)$.

Proof.

Let $\alpha = \{\alpha_1, \alpha_2, \ldots \alpha_n\}$, $\alpha' = \{\alpha_1', \alpha_2', \ldots \alpha_k'\}$ such that $\alpha \leq \alpha'$. By Proposition 3.3.1 we have for any $\alpha_i' \in \alpha'$ there exists $\alpha_j \in \alpha$: $\alpha_i' \subseteq \alpha_j$. Thus, for any $\sigma \in T^{\infty}(\Sigma)$ we have α_i' enabled implies α_j enabled, and α_i' taken implies α_j taken. Hence, α_i' enabled infinitely often implies α_j enabled infinitely often. Thus, if σ is $SPF(\Pi')$ then, by weakening the premise, we have $SPF(\Pi)$.

The remaining implications can be proved similarly.

❑

Thus, a chain of refinements of a given alphabet structure gives rise to a decreasing chain of classes of computations of a given ATS that are fair with respect to the given fairness notion.

Relationship to Maximality and Event Fairness

It is easy to see that weak process fairness with respect to minimal process structure $\Pi^{min} = (\pi, \alpha^{min})$ is the weakest possible process fairness notion. This notion is also called *concurrency fairness* because it guarantees the progress of concurrent *and* independent components without imposing any restrictions on the conflicts present in the system, whether these conflicts are due to synchronisation or internal choice. Concurrency fairness corresponds to justice [LPS81].

It should be stressed here that when using interleaving semantics, for example, as a labelled transition system, assumptions like justice have to be added to exclude those behaviours that, although allowed by the underlying model, do not respect concurrency in the sense that would be automatically guaranteed by a truly concurrent implementation. However, when using trace semantics, the additional information in the form of independency that is available in the underlying model allows a straightforward selection of the class of *admissible* computations. It turns out that the class $WPFair(\Sigma,\Pi^{min})$ is exactly equal to the set of all admissible computations $Adm(\Sigma)$ of the given system Σ, namely $(Max(T^{\infty}(\Sigma))) \cup T^{*}(\Sigma)$. This is one of the benefits we can reap when using non-interleaving semantics to model concurrency.

Unconditional process fairness with respect to maximal process structure $\Pi^{max} = (\pi, \alpha^{max})$ is the strongest possible process fairness notion. Not surprisingly, it coincides with event fairness as defined in previous section.

We summarize the results in the following theorem.

Proposition 4.3.3.
Let $S = (Q, A, \rightarrow, \iota)$ be an ATS, $\Sigma = (S, q_0)$ be a rooted ATS, and let $\Pi^{min} = (\pi, \alpha^{min})$, $\Pi^{max} = (\pi, \alpha^{max})$ be the minimal and maximal process structures over Σ respectively. Then:

(i) $WPFair(\Sigma,\Pi^{min}) = Adm(\Sigma) = Max(T^{\infty}(\Sigma)) \cup T^{*}(\Sigma)$

(ii) $WPFair(\Sigma,\Pi^{max}) = WEFair(\Sigma)$

(iii) $WEFair(\Sigma) \subseteq WPFair(\Sigma,\Pi)$, for any process structure Π over Σ.

Proof.
(i) Obviously, the equality holds for finite computations. We first show $Adm(\Sigma) \subseteq WPFair(\Sigma,\Pi^{min})$ (proof by contradiction). Let $\sigma \in T^{\omega}(\Sigma)$ such that σ is *not* $WPF(\Sigma,\Pi^{min})$. Then, from definition of WPF, $\exists\, \alpha_k \in \alpha^{min}, \exists\, \tau \in Pref^{fin}(\sigma)$ such that α_k continuously enabled in σ/τ and α_k not taken in σ/τ. Thus, by definition of α_k continuously enabled in σ/τ we have $\exists\, \alpha_k \in \alpha^{min}, \exists\, \tau \in Pref^{fin}(\sigma)$ such that $(\forall\, \gamma \in Pref^{fin}(\sigma/\tau)\ \exists\, a \in \alpha_k\colon\ \tau\gamma[a] \in T^{*}(\Sigma)) \wedge (p_k(\sigma/\tau) = \varepsilon)$. Since α_k is not taken in σ/τ, it may be concluded that $b \notin \alpha_k$ for all $b \in Act(\sigma/\tau)$. Thus, because $\alpha_k \in \alpha^{min}$ and $a \in \alpha_k$ it follows that a is independent of all $Act(\sigma/\tau)$. We have thus shown that $\exists\, \tau \in Pref^{fin}(\sigma)$ such that $(\forall\, \gamma \in Pref^{fin}(\sigma/\tau)\ \exists\, a \in \alpha_k\colon\ \tau\gamma[a] \in T^{*}(\Sigma)) \wedge (a\ \iota\ Act(\sigma/\tau))$. Thus, by Theorem 3.5.11, σ is not admissible. This concludes the proof.

We show $WPFair(\Sigma,\Pi^{min}) \subseteq Adm(\Sigma)$ (proof by contradiction). Let $\sigma \in T^{\omega}(\Sigma)$ such that σ is *not* in $Adm(\Sigma)$; hence $\sigma \notin Max(T^{\infty}(\Sigma))$. Then, by Theorem 3.5.11, $\exists\, \gamma \in Pref^{fin}(\sigma), \exists\, a \in A\colon\ \gamma[a] \in T^{*}(\Sigma) \wedge (a\ \iota\ Act(\sigma/\tau))$. From condition (i) of the definition of ATS it follows that for every $\gamma' \in Pref^{fin}(\sigma)$ such that $\gamma \leq \gamma'$ we have $\gamma'[a] \in T^{*}(\Sigma)$. We have thus shown that σ is *not* in $WPFair(\Sigma,\Pi^{min})$, which concludes the proof.

(ii) The result follows directly from the fact that α^{max} must, by definition, include all singleton sets $\{a\}$ for all $a \in A$.

(iii) This is a consequence of (i), together with the fact that the maximal process

structure is a refinement of any other process structure over the same concurrent alphabet.

□

Example. Let us consider trace semantics of the Condition/Event net shown in Fig. 4.2.2. A variety of alphabet structures are possible here, for example:

$\alpha_1 = \{\{a,b,d,e\},\{a,b,c\},\{d,e,f,g\}\}$

$\alpha_2 = \{\{a,b,d,e\},\{a,b,c\},\{d,e,f,g\}, \ldots, \{d\},\{g\}\}$.

α_1 is the minimal alphabet structure. Let $\Pi_1 = (\pi, \alpha_1)$, $\Pi_2 = (\pi, \alpha_2)$. The (admissible) computations $[(abc(abc)^*gf(gf)^*)^\omega]$ are UPF(Π_1), SPF(Π_1) and WPF(Π_1) because all three processes are taken infinitely often. The computation $[(abc)^\omega]$ is *not* admissible because it is not maximal, i.e. its every prefix can be extended with g. Note that $[(abc)^\omega]$ is not WPF(Π_1) either because the action g of the process {d,e,f,g} remains enabled continuously. The (admissible) computations $[(abc(abc)^*def(def)^*gf(gf)^*)^\omega]$ are clearly UPF(Π_1), SPF(Π_1), and WPF(Π_1). On the other hand, the computations $[(abc(abc)^*gf(gf)^*)^\omega]$ are *not* UPF(Π_2), SPF(Π_2) because the process {d}, although infinitely often enabled, is never taken. They are, however, WPF(Π_2) because {d} is not continuously enabled. The computation $[(abc)^\omega]$ is not admissible, nor is it WPF(Π_2). The (admissible) computations $[(abc(abc)^*def(def)^*gf(gf)^*)^\omega]$ are SPF(Π_2) and WPF(Π_2) because all five processes are taken infinitely often.

4.4. Summary

We have introduced definitions of weak, strong and unconditional event and process fairness which, we believe, are suitable for non-interleaving concurrency. The presented formalism, however, does not express every possible notion of fairness; for example *equifairness* (see Chapter 2 for definition and overview), that is a fairness notion which takes into account relative frequency of conflict resolutions in favour of each transition in a group of simultaneously enabled transitions, is neither weak, strong, nor unconditional event fairness.

As an example, let us consider the asynchronous system $S = (Q, A, \rightarrow, \iota)$ shown in Fig. 4.4.1, where $\iota = \varnothing$.

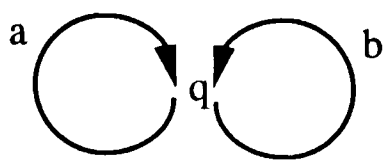


Fig. 4.4.1. An example of an ATS ($\iota = \varnothing$).

and a rooted ATS $\Sigma = (S, q)$. Actions a and b are simultaneously (and permanently) enabled in Σ. Equifairness would, apart from finite computations, include all such computations $\sigma \in \Theta_\iota{}^\omega$ for which there exists an infinite number of finite prefixes γ containing an equal number of occurrences of a's and b's. The following might be a definition of equifairness using projections. Let $EQF(\Sigma)$ denote the set of all computations of Σ which are equifair. Then:

$$EQF(\Sigma) = T^*(\Sigma) \cup \{\sigma \in T^\omega(\Sigma) \mid \mathit{eqf}(\sigma)\}$$

where $\mathit{eqf}(\sigma)$ is a predicate which holds iff:

$$\mathrm{card}\{\gamma \in \mathrm{Pref}^{\mathrm{fin}}(\sigma) \mid |x/\{a\}| = |x/\{b\}| \text{ for some } x \in \gamma\} = \omega.$$

$EQF(\Sigma)$ will, for example, contain $[(ab)^\omega]$, but not $[a(ab)^\omega]$. Note that this does not coincide with weak, strong and unconditional event fairness since:

$$WEF(\Sigma) = SEF(\Sigma) = UEF(\Sigma) = T^*(\Sigma) \cup \{\sigma \in T^\omega(\Sigma) \mid (|x/\{a\}| = \omega) \wedge (|x/\{b\}| = \omega)\}$$

which would contain both $[(ab)^\omega]$ and $[a(ab)^\omega]$.

Summarizing, what we have presented in this chapter is a subclass of fairness properties that are determined by *alphabet structures* over a given concurrent alphabet, allowing only a restricted set of *strength* predicates. Rather than look for more predicates and build fairness notions corresponding to them, we now attempt to provide a definition of an *abstract* fairness notion, not necessarily relative to alphabet structures and the class of strength predicates introduced so far.

5

Mathematical Space of Behavioural Properties

A concurrent alphabet (A, ι) determines the set of all traces $(\Theta_\iota{}^\infty, \leq_\iota)$, which forms a domain with trace prefix ordering. The domain $(\Theta_\iota{}^\infty, \leq_\iota)$ constitutes an abstraction of the class of all computations of asynchronous transition systems over the given alphabet, and is thus referred to as the computation space. Asynchronous transition systems are abstract representations of concurrent and non-deterministic programs, or systems, while infinitary trace languages correspond to their behaviours. Since the emphasis of our work is on the behavioural aspects of systems, rather than their input-output relationship, we can view infinitary trace languages as specifications. A specification of a program is a (finite or countable) list of properties the program is to satisfy. The program is then said to satisfy the specification if it satisfies each property.

The purpose of this chapter is to provide a mathematical space of system properties over the computation space $(\Theta_\iota{}^\infty, \leq_\iota)$ determined by the concurrent alphabet (A, ι). As suggested in [Smy83], a mathematical space of system properties over a cpo structure can be conveniently introduced through a topological characterization. We distinguish two commonly recognised classes of properties, namely *safety* and *progress* properties, which we formally define. We then define *fairness properties* as a subclass of infinitary progress properties closed under union. Within the class of fairness properties, we distinguish process and event fairness properties. We show that a variety of process fairness properties could be built given a concurrent alphabet and strength predicates. Process fairness properties form a lattice with inclusion ordering. The relationship of fairness and progress properties is also investigated.

5.1. Defining Properties

Properties are usually split into two classes [Lam77] [Pnu86] [AlS85], namely safety, also called invariance properties, and liveness properties. The nomenclature is motivated by different proof techniques used in each case. While the verification of safety properties is based on the invariance argument, liveness requires the use of well-founded sets.

Informally, a safety property states that *nothing bad will happen* in the course of computation. Examples of safety properties are *absence of deadlock*, where the "bad thing" is the program reaching deadlock, and *mutual exclusion*, where it refers to two concurrent processes simultaneously executing their critical sections. A safety property does not guarantee that some state will eventually be reached during program execution, only that it will *always* be the case that none of the "unsafe" states will be visited.

This is why it is necessary to distinguish liveness, that is a property which informally states that *something good will happen* in the course of computation. Examples of liveness are *program termination*[1], where the "good thing" is the program reaching the final state, and *guaranteed response*, where the "good thing" is the request for service being granted. Unlike safety, liveness properties are useful when showing that some "good" thing will *eventually* happen.

Fairness is usually viewed as a liveness property [Pnu86], but it is also possible to consider fairness as a constraint [Par85], i.e. a set that specifies those infinite computations that need to be excluded. The distinguishing feature of an unfair computation is the lack of progress of some component of the system, which is exhibited only by infinite computations (all finite computations are considered fair). This gives fairness a distinct infinitary flavour.

We shall now attempt to define each of the above-mentioned classes of properties in the non-interleaving framework represented by the domain $(\Theta_\iota^\infty, \leq_\iota)$. Note that, although traces contained in Θ_ι^∞ are equivalence classes of sequences of action labels, they can also

---

1 Sometimes liveness includes a safety component, for example *total correctness* would be an example of a liveness property in this approach.

be used to uniquely identify (global) system states. This is because two equivalent derivations determine the same state in an unambiguous rooted asynchronous transition system (Proposition 3.6.1). Trace prefix ordering may now be viewed as a partial order relation on the states visited during system execution. Incomparable prefixes of the same trace correspond to states arising due to concurrent execution, while inconsistent traces are a result of conflict resolution (non-determinism).

Motivation and Background

It has been noted in [Smy83] that a topological space X should be viewed as a "data type", with the open sets as the (computable) properties defined on that type. Given a space X, let us define a predicate on the space X to be a continuous map from X into the Boolean cpo B = $\{\bot \leq ff,\ \bot \leq tt\}$. Taking the Scott topology over B, we have $\varnothing$, {ff}, {tt} and $\{\bot, ff, tt\}$ as the open sets. Since {tt} is open, then $p^{-1}(tt)$ will be open (by topological definition of a continuous map). Conversely, suppose S is open. Define a function p by:

$$p(x) = \begin{cases} tt & x \in S \\ \bot & x \notin S. \end{cases}$$

Note that $p^{-1}(tt) = S$, $p^{-1}(ff) = \varnothing = p^{-1}(\varnothing)$, $p^{-1}(\{\bot, ff, tt\}) = \{\bot, ff, tt\}$. Since all sets are open, we have p is continuous. Hence, a subset S of X is open if, and only if, S is $p^{-1}(tt)$ for some predicate p.

The above justifies why properties may be identified with the open sets. Introducing a topology over a domain is therefore a convenient way of providing an algebra of properties that is closed under union and intersection. Since the set of all traces $(\Theta_\iota^\infty, \leq_\iota)$ forms a domain containing the behaviours of all asynchronous transition systems over the same concurrent alphabet, one might pose a question if it is possible to utilize topological ideas in order to provide a mathematical space of *behavioural* properties of such systems.

At this point, several questions spring to mind. Apart from the obvious pragmatic problem of how one chooses the subsets of Θ_ι^∞ so that they represent useful properties, it would seem sensible to allow those properties only that are computable in some sense. A reasonable notion of computability, as suggested in [Smy83], would be that in which property P is considered computable if, and only if, the set of codes of (computable)

elements satisfying P can be effectively enumerated. Intuitively, the idea of a computable property is that there exists a uniform procedure for a property P that for any (finite) element x tells us in finite time that P(x) holds, whenever this is true (semi-decidability). Another important point made in [Smy83] is that computability concepts are relative to the open bases chosen, i.e. properties should be represented as open basic sets and their unions. In our setting, this would imply that a computable base should be determined by finite elements of the domain $(\Theta_\iota{}^\infty, \leq_\iota)$.

A specification of a program is a countable list of properties that the program is to satisfy. Since properties are identified with open sets, this means that what is specified is always a countable intersection of a family of open sets. A topological notion that exactly corresponds to the above is a G_δ-set, that is a countable (finite or infinite) intersection of a family of open sets.

Notation

All definitions and notation conventions used in this chapter have been introduced in Section 3.1. The following is a short summary of the notation adopted for the purpose of this chapter. Let $(D, \leq)$ be a domain. For any subset X of we define the *prefix closure* of X, denoted $\downarrow X$, as $\{y \in D \mid \exists\, x \in X:\ y \leq x\}$ (the lower set). We shall also distinguish closure with respect to finite prefixes, denoted $\downarrow^{fin} X$, which is defined as $X \cup \{y \in B_D \mid \exists\, x \in X:\ y \leq x\}$. The *suffix closure* of X, denoted $\uparrow X$, is defined by $\{y \in D \mid \exists\, x \in X:\ x \leq y\}$ (the upper set). $\downarrow\{x\}$, $\uparrow\{x\}$ and $\downarrow^{fin}\{x\}$, for $x \in D$, are abbreviated to $\downarrow x$, $\uparrow x$ and $\downarrow^{fin} x$ respectively.

Properties and Satisfaction

A *property* is defined as a subset Ψ of the computation space $\Theta_\iota{}^\infty$ over the concurrent alphabet (A, ι). A property $\Psi \subseteq \Theta_\iota{}^\infty$ is an *infinitary* property if, and only if, $\Psi \subseteq \Theta_\iota{}^\omega$, and *finitary* otherwise.

It should be noted here that our definition differs from the usual definition introduced in the interleaving models, where often a property is defined as a set of sequences of system states [AlS85], which, clearly, is equivalent to defining it as a subset of A^∞ for

unambiguous rooted systems. For a subset Ψ of A^∞ to correspond to a property in our sense we would require Ψ to be closed under trace equivalence. This is without loss of generality as by Proposition 3.6.1 any two equivalent derivations of a rooted unambiguous ATS determine the same state.

Let $S = (Q, A, \rightarrow, \iota)$ be an ATS, $\Sigma = (S, q_0)$ be a rooted ATS over S, $q_0 \in Q$. A trace $\sigma \in \Theta_\iota{}^\infty$ *has the property* Ψ if σ is contained in Ψ. Σ is said to *satisfy* property Ψ if, and only if, the set of all admissible computations of Σ is contained in $\downarrow^{fin}\Psi$, that is:

$$Adm(\Sigma) \subseteq \downarrow^{fin}\Psi,$$

where the usual definition of admissibility applies, namely:

$$Adm(\Sigma) = T^*(\Sigma) \cup Max(T^\infty(\Sigma)).$$

Note that it is possible for a system to terminate *before* reaching a trace in Ψ. Of course, a trivial ATS whose sole computation is $[\varepsilon]$ satisfies any non-empty property.

We require that all finite computations $T^*(\Sigma)$ of Σ are contained in $\downarrow^{fin}\Psi$, but of infinite computations only those computations that are maximal in Σ must be contained in Ψ. This is sufficient because all finite computations of Σ are approximations of the set $Max(T^\infty(\Sigma))$. We impose this constraint in order to guarantee that only global, objective computations of the system behaviour are considered.

Example. Fig. 5.1.1 shows the computation space D_1 determined by the concurrent alphabet (A, ι) with $A = \{a,b\}$ and $a \iota b$. Examples of properties over D_1 are $\Psi_1 = [a^*b^*] \cup [(a^*b^*)^\omega]$, which is a finitary property, and $\Psi_2 = \{[(ab)^\omega]\}$, which is infinitary. The rooted asynchronous system Σ shown in Fig.5.1.3 satisfies Ψ_1 and Ψ_2. Examples of properties that are not satisfied by Σ are $\Psi_3 = \{[ab]\}$ and $\Psi_4 = [a^*] \cup \{[a^\omega]\}$.
Let us also consider the computation space D_2, where $\iota = \varnothing$, shown in Fig. 5.1.2. Examples of properties over D_2 are $\Psi_5 = [(a^*b^*)^*] \cup [(a^*b^*)^\omega]$, $\Psi_6 = [(a^*b^*)^*] \cup \{[(aa^*bb^*)^\omega]\}$. When considering Σ over D_2 we have Σ satisfies Ψ_5, but does not satisfy Ψ_6.

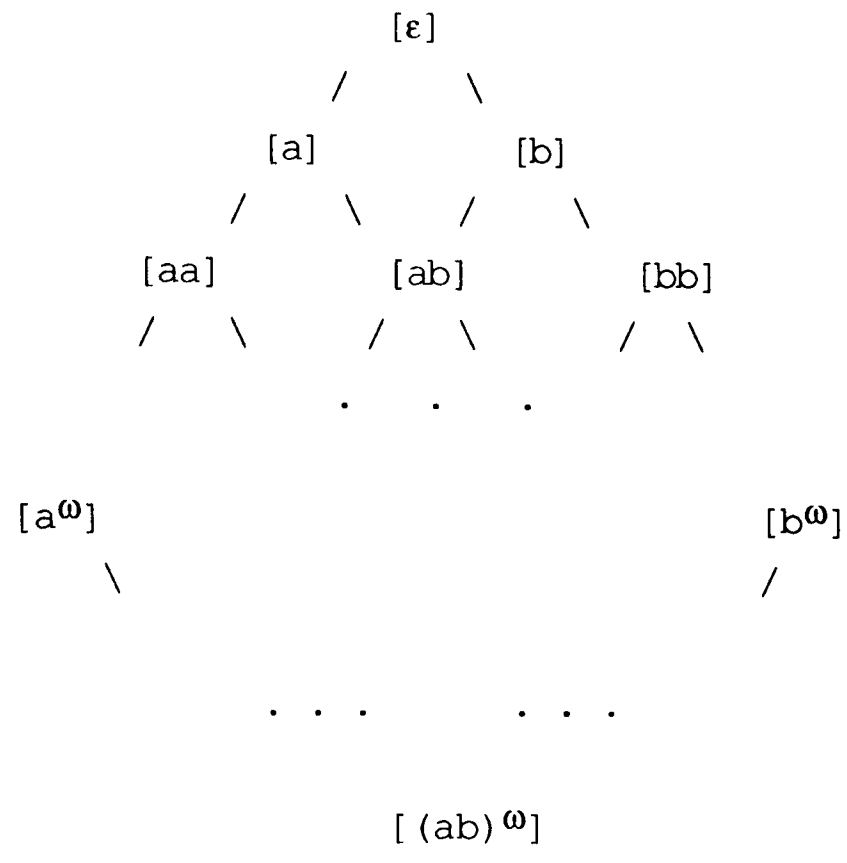


Fig. 5.1.1. The domain D_1 of traces determined by A = {a,b} with a ι b.

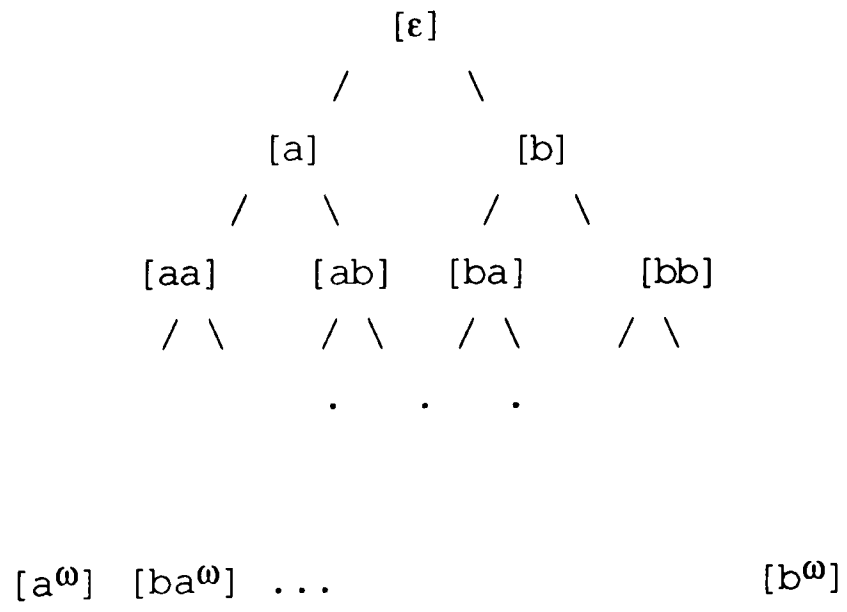


Fig. 5.1.2. The domain D_2 of traces determined by A = {a,b} with ι = ∅.

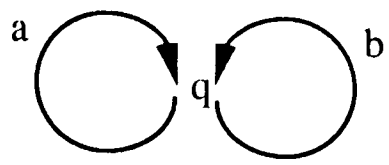


Fig. 5.1.3. A rooted ATS $\Sigma = (S, q)$.

Example. Our definition of a property differs from the usual definition used in interleaving models. Let us again consider the domains D_1 and D_2 shown in Fig. 5.1.1 and Fig. 5.1.2. Note that $S = \{ab\}$ is not a property in D_1 (because it is not closed under trace equivalence - it would need to contain the string ba as well), but it is a property in D_2 (because the string ba is not equivalent to ab).

Safety Properties

A property Ψ is a *safety property* if, and only if, Ψ is prefix-closed, that is, $\Psi = \downarrow\Psi$. This definition states that if a computation, possibly infinite, has a given property then *every* approximation of this computation has this property. Also, if some computation is not in the property, then none of the extensions of this computation can be in the property[2]. Every computation in a safety property represents the invariant "so far nothing bad has happened". The role of the system is to restrict the computation space by evolving only through the "safe" moves, in which case we can conclude that absence of the "bad thing" is *always* the case.

The class of safety properties forms a topology $\mathcal{T}_S$ (of Alexandroff-open sets for the converse of the trace prefix ordering relation). The open base of this topology is $\mathcal{B}_S = \{\downarrow\sigma \mid \sigma \in \Theta_\iota^\infty\}$ where σ is not necessarily a finite element. The closure operation for $\sigma \in \Theta_\iota^\infty$ is defined as $\uparrow\sigma$. The closure of a subset Y of Θ_ι^∞ is thus $\uparrow Y$. The closed sets in this topology, namely the Alexandroff topology for the converse of trace prefix ordering, are suffix closed sets ($\Psi = \uparrow\Psi$), or, in other words, Alexandroff-open sets with respect to trace prefix ordering.

---

2 This corresponds to the usual characterization of safety as "once lost, they can never be regained" [Pnu86].

Safety properties are not necessarily closed infinitary trace languages.

Proposition 5.1.1.

Suffix closure satisfies axioms 1-4 of topological closure.

Proof.

1) Let $X, Y \subseteq \Theta_1^{\infty}$. Proof that $\uparrow(X \cup Y) = \uparrow X \cup \uparrow Y$ is standard.

2) $X \subseteq \uparrow X$ follows from reflexivity of $\leq_1$.

3) $\uparrow\varnothing = \varnothing$ follows from definition of suffix closure.

4) $\uparrow(\uparrow X) = \uparrow X$ is obvious.

Axiom 5 is not satisfied, hence $\mathcal{T}_S$ is not $\mathcal{T}_1$.

❑

Proposition 5.1.2.

(i) $\varnothing$, $\{[\varepsilon]\}$, Θ_1^*, Θ_1^{∞} are safety properties.

(ii) If Ψ_i is a countable family of safety properties then $\cup\Psi_i$ is a safety property.

(iii) If Ψ_i is a countable family of safety properties then $\cap\Psi_i$ is a safety property. $\cap\Psi_i$ is non-empty iff for all i, Ψ_i is non-empty.

Proof.

(i) Obvious.

(ii) Suppose Ψ_i is a countable family of safety properties, then from definition we have for each i, $\Psi_i = \downarrow\Psi_i$. Clearly, $\cup\Psi_i = \downarrow(\cup\Psi_i)$.

(iii) Suppose Ψ_i is a countable family of safety properties. Clearly, $\cap\Psi_i = \downarrow(\cap\Psi_i)$. The second part follows from the fact that every non-empty safety property contains $[\varepsilon]$.

❑

Example. Examples of safety properties over the computation space D_1 shown in Fig.5.1.1 are:

$\Psi_1 = \{[\varepsilon]\} \cup [ab^*] \cup \{[ab^{\omega}]\}$ "always a at most once"

$\Psi_2 = [a^*] \cup \{[a^{\omega}]\}$. "always a"

Sample safety properties over D_2 shown in Fig. 5.1.2 are:

$\Psi_3 = \{[\varepsilon]\} \cup [a(a^*b^*)^*] \cup [a(a^*b^*)^{\omega}]$, "always a precedes b"

$\Psi_4 = [a^*] \cup \{[a^{\omega}]\} \cup [b^*] \cup \{[b^{\omega}]\}$, "always a or always b"

$\Psi_5 = [a^*b^*] \cup [(a^*b^*)^{\omega}]$. "always a or b"

Note that $\Psi_6 = \{[ab]\}$ and $\Psi_7 = \{[(ab)^{\omega}]\}$ are not safety properties.

The ATS shown in Fig. 3.3.3 satisfies Ψ_1 but it does not satisfy Ψ_2. The ATS shown in Fig. 3.3.6 over the computation space included in Fig. 5.1.2 satisfies Ψ_5 and Ψ_3, but it does not satisfy Ψ_4 (assume b stands for τ_b).

Progress Properties

A property Ψ is a *progress property* if, and only if, it is suffix-closed ($\Psi = \uparrow\Psi$), i.e. it is Alexandroff-open. This definition of the progress property states that, as soon as the required progress has been made, it is irrevocable, which is expressed by the fact that all extensions of computations in the given property must be included as well[3]. In contrast to safety properties, it is feasible that some, possibly infinite, computation is contained in a progress property, but none of its proper finite prefixes are included. A progress property cannot be avoided, but it is possible for the system to terminate before a computation with this property has been reached.

Progress properties correspond to liveness [Lam77] [AlS85] [Pnu86]. They form a topology over a cpo, Alexandroff topology, which we shall denote $\mathcal{T}_P$. The base of this topology is $\mathcal{B}_P = \{\uparrow\sigma \mid \sigma \in \Theta_\iota^\infty\}$, where σ is not necessarily a finite element. The closure operation for $\sigma \in \Theta_\iota^\infty$ can be defined as $\downarrow\sigma$ (prefix closure). The closure of $Y \subseteq \Theta_\iota^\infty$ is thus $\downarrow Y$. The closed sets in this topology are prefix-closed sets ($\downarrow Y = Y$).

Note that progress properties are not, in general, closed nor prefix-closed infinitary trace languages.

Proposition 5.1.3.
Prefix closure satisfies axioms 1-4 of topological closure.
Proof.
1) Let $X, Y \subseteq \Theta_\iota^\infty$. Proof that $\downarrow(X \cup Y) = \downarrow X \cup \downarrow Y$ is standard.
2) $X \subseteq \downarrow X$ follows from reflexivity of $\leq_\iota$.
3) $\downarrow\varnothing = \varnothing$ follows from definition of prefix closure.
4) $\downarrow(\downarrow X) = \downarrow X$ is obvious.
Axiom 5 is not satisfied, hence $\mathcal{T}_P$ is not $\mathcal{T}_1$.

❑

---

3 This corresponds to a statement "once gained, a progress property may never be lost".

Progress properties are closed under arbitrary union and arbitrary intersection.

Proposition 5.1.4.

(i) $\varnothing$, Θ_l^{∞}, Θ_l^{ω}, $\mathrm{Max}(\Theta_l^{\infty})$ are progress properties.

(ii) If Ψ_i is a countable family of progress properties then $\cup\Psi_i$ is a progress property.

(iii) If Ψ_i is a countable family of progress properties then $\cap\Psi_i$ is a progress property.

Proof.

(i) Obvious.

(ii) Suppose Ψ_i is a countable family of progress properties, then from definition we have for each i, $\Psi_i = \uparrow\Psi_i$. Clearly, $\cup\Psi_i = \uparrow(\cup\Psi_i)$.

(iii) Suppose Ψ_i is a countable family of progress properties. Clearly, $\cap\Psi_i = \uparrow(\cap\Psi_i)$.

❑

Example. Examples of progress properties over the domain D_1 shown in Fig. 5.1.1 are:

$\Psi_1 = \{[a^{\omega}],[ba^{\omega}], [bba^{\omega}], \ldots [(ab)^{\omega}]\}$, "eventually inf.often a"

$\Psi_2 = \{[bb]\}\cup[a^*bbb^*]\cup\{[bba^{\omega}], \ldots [(ab)^{\omega}]\}$, "eventually bb"

$\Psi_3 = [(ab)^{\omega}]$. "eventually inf.often a and b"

The system shown in Fig. 5.1.3 considered over D_1 satisfies Ψ_1, Ψ_2 and Ψ_3.

Examples of progress properties over D_2 included in Fig. 5.1.2 are:

$\Psi_4 = \{[(aa^*b^*)^{\omega}]\}$, "eventually inf.often a"

$\Psi_5 = [(a^*bbb^*)^*] \cup [(a^*bbb^*)^{\omega}]$, "eventually bb"

$\Psi_6 = [(bb^*aa^*)^{\omega}]$, "eventually inf.often a and b"

Properties which are not progress properties in D_2 are $\Psi_7 = \{[a]\}$ and $\Psi_8 = [(ab)^*] \cup \{[(ab)^{\omega}]\}$ as they are not suffix-closed. The system in Fig.5.1.3 over D_2 does not satisfy Ψ_4, Ψ_5 and Ψ_6.

Infinitary Progress and Computability

We have deliberately relaxed the definition of properties to allow for infinitary properties although, in the sense of [Smy83], infinitary properties are not computable. We can,

however, show that every infinitary progress property is an intersection of a countable family of finitary progress properties (the latter class are computable).

Let us consider infinitary progress properties, that is progress properties Ξ such that $\Xi \subseteq \Theta_\iota{}^\omega$.

Proposition 5.1.5.
Every infinitary progress property is an intersection of a countable family of finitary progress properties.

Proof.
Suppose $\Xi \subseteq \Theta_\iota{}^\omega$ is an infinitary progress property, and define a countable family $\uparrow\Psi_i$, for $i \in N$, by taking:

$$\Psi_i = \downarrow^{fin}\Xi \cap \{\sigma \in \Theta_\iota{}^* \mid |\sigma| = i\}.$$

Note that for each i, $\uparrow\Psi_i$ is a finitary progress property. It is clear that $\Xi = \cap(\uparrow\Psi_i)$.

□

It is easy to see that finitary progress properties are exactly Scott-open sets. A set $\Psi \subseteq \Theta_\iota{}^\infty$ is Scott-open if, and only if, it is suffix-closed ($\Psi = \uparrow\Psi$) and for every directed set M $\subseteq \Theta_\iota{}^\infty$, if lub(M) is in Ψ then some element of M must be in Ψ. It is a standard result [GHK80] that Scott-open sets form a topology over a cpo, which we shall denote $\mathcal{T}\mathrm{P}^{fin}$ (the topology of finitary progress properties). The base of this topology is $\mathcal{B}\mathrm{P}^{fin} = \{\uparrow\sigma \mid \sigma \in \Theta_\iota{}^*\}$, that is, σ is a finite element. Every basic finitary progress property is thus computable. Finitary progress properties are closed under arbitrary union and finite intersection.

Clearly, infinitary progress properties are G_δ-sets with respect to the topology $\mathcal{T}\mathrm{P}^{fin}$ of finitary progress properties.

Let $Y \subseteq \Theta_\iota{}^\infty$. The closure operation in the topology $\mathcal{T}\mathrm{P}^{fin}$ is $Cl(Y) = \downarrow Y \cup Adh(\downarrow Y)$.

Proposition 5.1.6.
Closure operation satisfies axioms 1-4 of topological closure.
Proof.
1) Let $X, Y \subseteq \Theta_\iota{}^\infty$. Proof that $\downarrow(X \cup Y) = \downarrow X \cup \downarrow Y$ is standard. $Adh(\downarrow(X \cup Y)) = Adh(\downarrow X) \cup Adh(\downarrow Y)$ is clear.
Axioms 2, 3, 4 are obvious.

Axiom 5 is not satisfied, hence $\mathcal{T}_P^{fin}$ is not $\mathcal{T}_1$.

❑

Relationship of Safety and Progress

Safety properties are always finitary; although safety properties may contain some infinite traces, this is only the case if all finite prefixes of this trace are contained in the property. Thus, in order to verify a safety property it is sufficient to show that it holds initially and that it is preserved by every transition of the program (this is the essence of the invariance argument).

On the other hand, purely infinitary progress properties are possible. This complicates the verification process as no argument based on the analysis of locally reachable computations (states) can show in finite time that a given infinitary progress property holds.

A further difference between the two kinds of properties is the following. A system satisfies a safety property if, and only if, all its admissible computations are contained in the property. For a system to satisfy a progress property, all its admissible finite computations must be finite approximations of some, possibly infinite, computation contained in the property. In other words, a safety property specifies all "safe" computations the system must take, while a progress property specifies the "goal" without imposing any restrictions on how the system chooses the computations in order to meet the required goal; the only restriction is that the system evolves through computations that consist of finite steps.

It is clear that progress properties are the closed sets with respect to the topology $\mathcal{T}_S$ of safety properties. Safety properties are the closed sets in the topology $\mathcal{T}_P$ of progress properties. Thus, the complement of a safety property is always a progress property and, vice-versa, the complement of a progress property is always a safety property.

Proposition 5.1.7.
Progress properties are exactly the closed sets in the topology of safety properties $\mathcal{T}_S$.
Safety properties are exactly the closed sets in the topology of progress $\mathcal{T}_P$.

Proof.
The closure operation with respect to $\mathcal{T}_S$ is suffix closure; since progress properties are, by definition, suffix-closed, they are exactly the closed sets with respect to $\mathcal{T}_S$.
Similarly, the closure operation with respect to $\mathcal{T}_P$ is prefix closure; hence, safety

properties are the closed sets in $\mathcal{T}_P$.

❑

It is an interesting question if the topology $\mathcal{T}_P^{fin}$ of finitary progress properties has a counter-part in terms of safety properties. The answer to this turns out to be positive. A subset of safety properties in $\mathcal{T}_S$, namely those safety properties Ψ that are closed infinitary trace languages (i.e. $Adh(\Psi) \subseteq \Psi$), are exactly the closed sets with respect to the topology $\mathcal{T}_P^{fin}$.

Many properties are neither safety nor progress, but every property is contained in the intersection of some safety and some progress property. Unfortunately, unlike in [AlS85], it is not always the case that the given property is exactly the intersection of the two properties.

Proposition 5.1.8.
Let Ψ be a property in Θ_ι^∞, then there exists the least safety property S and the least progress property P such that:

$$\Psi \subseteq S \cap P.$$

Proof.
Let Ψ be a property in Θ_ι^∞. For every $\sigma \in \Psi$ let us define:

$$S_\sigma = \downarrow\sigma$$
$$P_\sigma = \uparrow\sigma.$$

Clearly, $S = \cup\{S_\sigma \mid \sigma \in \Psi\}$ is a safety property and $P = \cup\{P_\sigma \mid \sigma \in \Psi\}$ is a progress property. It follows from construction that $\Psi \subseteq S \cap P$ and S, P are the least safety and progress properties containing Ψ.

❑

Example. Properties which are neither safety nor progress in the computation space D_1 shown in Fig.5.1.1 are $\Psi_1 = [ab]$ and $\Psi_2 = [a^*b^*] \cup [(ab)^\omega]$. Note that $\Psi_1 = \downarrow[ab] \cap \uparrow[ab]$, but $\Psi_2 \subseteq \downarrow\Psi_2 \cap \uparrow\Psi_2$.

5.2. Fairness and Progress Properties

Fairness Properties

Let $C = (A, \iota)$ be a concurrent alphabet. Fairness properties are a subclass of infinitary progress properties. A property $\Phi \subseteq \Theta_\iota{}^\omega$ is a *fairness property* if, and only if, Φ is a progress property (i.e. $\Phi = \uparrow\Phi$) such that $\Theta_\iota{}^* \subseteq \downarrow^{fin}\Phi$ if Φ is non-empty.

This is the weakest possible definition of a fairness property over a given concurrent alphabet. It states that every *finite* computation may be extended to an *infinite fair* computation, but some infinite computations may be excluded from it. Also, every extension of an infinite fair computation must be fair as well. This is consistent with the intuition about existing fairness notions.

There are infinitary progress properties which are not fairness properties; for such properties, there may well be finite computations which could not be extended to an infinite computation contained in the property. Fairness properties constitute a subset of the topology $\mathcal{T}_P$ closed under arbitrary union. In relation to the topology $\mathcal{T}_S$ of safety, fairness properties form a subset of the closed sets.

Fairness properties are not closed under intersection. The following is a counter-example (due to M.W. Shields). Let $A = \{a, b\}$ with $\neg (a\ \iota\ b)$ and define:

$$\Phi_1 = (a^*b^*)^*a^\omega \cup \{b^\omega\}$$
$$\Phi_2 = (a^*b^*)^*b^\omega.$$

Then $\Phi_1 \cap \Phi_2 = \{b^\omega\}$, which is not a fairness property.

Proposition 5.2.1.

(i) $\varnothing$, $\Theta_\iota{}^\omega$ are fairness properties.

(ii) $Max(\Theta_\iota{}^\infty)$ is a fairness property.

(iii) If Φ_i is a countable family of fairness properties then $\cup\Phi_i$ is a fairness property.

Proof.

(i) Obvious.

(ii) Follows from the fact that $\downarrow Max(\Theta_\iota^\infty) = \Theta_\iota^\infty$; thus $\Theta_\iota^* \subseteq \downarrow^{fin} Max(\Theta_\iota^\infty)$.
(iii) $\cup\Phi_i = \uparrow(\cup\Phi_i)$ follows from the fact that fairness properties are progress properties. It is clear that $\Theta_\iota^* \subseteq \downarrow^{fin}(\cup\Phi_i)$ whenever $\cup\Phi_i$ non-empty.

❑

Example. Examples of fairness properties over D_1 shown in Fig. 5.1.1 are:

$\Phi_1 = [(b^*a)^\omega]$, "inf.often a"

$\Phi_2 = [(a^*b)^\omega]$, "inf.often b"

$\Phi_3 = \{[(ab)^\omega]\}$. "inf.often a and b"

Sample fairness properties over D_2 shown in Fig. 5.1.2 are:

$\Phi_4 = [(b^*aa^*)^\omega]$, "inf.often a"

$\Phi_5 = [(a^*bb^*)^\omega]$, "inf.often b"

$\Phi_6 = [(aa^*bb^*a)^\omega]$, "inf.often a and b"

The system Σ shown in Fig. 5.1.3, when considered over D_1, satisfies fairness property Φ_1, Φ_2 and Φ_3. However, fairness properties Φ_4, Φ_5 and Φ_6 are not satisfied by Σ when considered over D_2.

An infinitary progress property in D_2 which is not a fairness property is $\{[a^\omega]\}$.

Relationship of Fairness and Progress

Fairness properties form a proper subclass of infinitary progress properties. All fairness properties, by definition, must allow for any finite computation to be extended to a fair computation. There are infinitary progress properties which are not fairness properties.

Fairness properties bear an interesting relationship with finitary progress properties. It turns out that fairness properties are the dense sets with respect to the topology $\mathcal{T}_P^{fin}$ of finitary progress properties. We summarize this observation in the proposition below.

Note that a similar statement cannot be shown for an arbitrary infinitary progress property.

Proposition 5.2.2.
Let $C = (A, \iota)$ be a concurrent alphabet, Φ a non-empty fairness property in Θ_ι^∞. Then Φ is dense in $\mathcal{T}_P^{fin}$.

Proof.

We need to show that $Cl(\Phi) = \downarrow\Phi \cup Adh(\downarrow\Phi) = \Theta_\iota^\infty$. From definition of fairness we have $\Theta_\iota^* \subseteq \downarrow^{fin}\Phi$, thus $\Theta_\iota^* \subseteq \downarrow\Phi$. Hence, $Adh(\downarrow\Phi) = \Theta_\iota^\omega$ and finally $Cl(\Phi) = \Theta_\iota^* \cup \Theta_\iota^\omega = \Theta_\iota^\infty$.

❑

Fairness and Processes

Process fairness properties form a subclass of fairness properties that are determined by alphabet structures. First, we show how unconditional process fairness may be defined in our formalism. For the time being we ignore strength predicates; we shall show in later sections how this restriction could be removed.

Let us consider the class of asynchronous transition systems with non-terminating processes. It may thus be concluded that, if a process has proceeded finitely often in an infinite admissible computation, then it must have been unfairly delayed while waiting to synchronise. An infinite computation is unconditionally process fair if, and only if, *every process proceeds infinitely often.*

Let $C = (A, \iota)$ be a concurrent alphabet, $\alpha \subseteq \wp(A)$ be an alphabet structure over C. We define $\Xi_u^{proc}(\alpha) \subseteq \Theta_\iota^\omega$ by:

$$\Xi_u^{proc}(\alpha) = \{\sigma \in \Theta_\iota^\omega \mid \forall\, \alpha_i \in \alpha:\ |p_i(\sigma)| = \omega\}.$$

A property $\Phi_u^{proc}(\alpha)$ is the *unconditional process fairness property with respect to* α if, and only if, it is the smallest fairness property containing $\uparrow\Xi_u^{proc}(\alpha) = \Xi_u^{proc}(\alpha)$.

This definition admits all finite computations together with all infinite computations which contain an infinite number of occurrences of actions from each of the alphabets of the agents. It is clear that $\cup\{\sigma \in \Phi_u^{proc}(\alpha)\}$ is a *fairmerge* [Par80] of string languages $(\alpha_i)^\infty$, $\alpha_i \in \alpha$.

Proposition 5.2.4. Let $C = (A, \iota)$ be a concurrent alphabet, $\alpha \subseteq \wp(A)$ be an alphabet structure over C. Then $\Phi_u^{proc}(\alpha)$ is well-defined.

Proof.
We show that $\Theta_l^* \subseteq \downarrow^{fin}\Phi_u^{proc}(\alpha)$. Suppose $\sigma \in \Theta_l^*$. We need to show that $\sigma \in \downarrow^{fin}\Phi_u^{proc}(\alpha)$, that is, there exists $\tau \in \Phi_u^{proc}(\alpha)$ such that $\sigma \leq_l \tau$. Since σ is finite, we have $\forall\ \alpha_i \in \alpha$: $p_i(\sigma)$ is a finite string. Take any γ such that $\forall\ \alpha_i \in \alpha$: $|p_i(\sigma)| = \omega$; then $\sigma \leq_l \sigma\gamma$ and $\forall\ \alpha_i \in \alpha$: $|p_i(\sigma\gamma)| = \omega$. Thus, we have constructed $\tau = \sigma\gamma \in \Phi_u^{proc}(\alpha)$ such that σ is a finite prefix of τ.

❑

It should be noted that the above fairness notions are not adequate when allowing for dynamic processes, that is recursion over concurrency.

Algebra of Unconditional Process Fairness

Unconditional process fairness properties form an algebra that is closed under arbitrary union and intersection. Since alphabet structures form a lattice with inclusion ordering and process fairness properties are determined by alphabet structures, it is not surprising that process fairness properties form a lattice with inclusion ordering. The ordering of unconditional process fairness properties corresponds to the refinement ordering of alphabet structures. A chain of refinements of a given alphabet structure gives rise to a chain of fairness notions of increasing strength. Intersecting alphabet structures corresponds to the weakening of process fairness properties, while taking the union of alphabet structures gives rise to a stronger fairness notion.

The following is a summary of results concerning unconditional process fairness.

Theorem 5.2.5. (Refinement Hierarchy of Unconditional Process Fairness)
Let $C = (A, \iota)$ be a concurrent alphabet, $\alpha, \alpha' \subseteq \wp(A)$ be alphabet structures over C.
(i) If α' is a refinement of α then $\Phi_u^{proc}(\alpha') \subseteq \Phi_u^{proc}(\alpha)$.
(ii) $\Phi_u^{proc}(\alpha) \cap \Phi_u^{proc}(\alpha') = \Phi_u^{proc}(\alpha \cup \alpha')$.
(iii) $\Phi_u^{proc}(\alpha) \cup \Phi_u^{proc}(\alpha') = \Phi_u^{proc}(\alpha \cap \alpha')$.

Proof.
(i) Suppose α' is a refinement of α, then by definition $\alpha \subseteq \alpha'$. Suppose $\Phi_u^{proc}(\alpha')$, $\Phi_u^{proc}(\alpha)$ are the smallest fairness properties containing $\uparrow\Xi_u^{proc}(\alpha)$ and $\uparrow\Xi_u^{proc}(\alpha')$ respectively, where $\Xi_u^{proc}(\alpha) = \{\sigma \in \Theta_\iota^\omega \mid \forall\ \alpha_i \in \alpha: |p_i(\sigma)| = \omega\}$. Suppose $\sigma \in \uparrow\Xi_u^{proc}(\alpha')$; then $\forall\ \alpha_i' \in \alpha'$: $|p_i(\sigma)| = \omega$. By Proposition 3.3.1 we have for each α_i'

$\in \alpha'$ there exists $\alpha_k \in \alpha$ such that $\alpha_i' \subseteq \alpha_k$. Since $|p_i(\sigma)| = \omega$, it follows that $|p_k(\sigma)| = \omega$ and thus $\sigma \in \uparrow\Xi_u^{proc}(\alpha)$.

(ii) $\Phi_u^{proc}(\alpha \cup \alpha') \subseteq \Phi_u^{proc}(\alpha) \cap \Phi_u^{proc}(\alpha')$ follows from (i) and the observation that $\alpha \cup \alpha'$ is a refinement of α (similarly, $\alpha' \cup \alpha'$ is a refinement of α).
We show $\Phi_u^{proc}(\alpha) \cap \Phi_u^{proc}(\alpha') \subseteq \Phi_u^{proc}(\alpha \cup \alpha')$. Suppose $\sigma \in \uparrow\Xi_u^{proc}(\alpha) \cap \uparrow\Xi_u^{proc}(\alpha')$, then $\forall\ \alpha_i \in \alpha$: $|p_i(\sigma)| = \omega$ and $\forall\ \alpha_k' \in \alpha'$: $|p_k(\sigma)| = \omega$. It is clear that $\forall\ \alpha_m \in \alpha \cup \alpha'$: $|p_m(\sigma)| = \omega$, which concludes the proof.

(iii) Proof similar to (ii).

❑

It is not difficult to see that the weakest unconditional process fairness property over a given concurrent alphabet is $\Phi_u^{proc}(\alpha^{min})$ where α^{min} is the minimal alphabet structure. It is the union of all unconditional process fairness properties over the given alphabet. On the other hand, $\Phi_u^{proc}(\alpha^{max})$, where α^{max} is the maximal alphabet structure, is the strongest possible unconditional process fairness property given a concurrent alphabet. $\Phi_u^{proc}(\alpha^{max})$ is the intersection of all process fairness properties with respect to the given concurrent alphabet.

Theorem 5.2.6. (Lattice of Unconditional Process Fairness)

(i) Unconditional process fairness properties form a lattice with inclusion.

(ii) $\Phi_u^{proc}(\alpha) \subseteq \Phi_u^{proc}(\alpha^{min})$ for any alphabet structure α.

(iii) $\Phi_u^{proc}(\alpha^{max}) \subseteq \Phi_u^{proc}(\alpha)$ for any alphabet structure α.

Proof.

This is a consequence of the above proposition and the fact that for any alphabet structure α, $\alpha^{min} \subseteq \alpha$ and $\alpha \subseteq \alpha^{max}$.

❑

Example. The only unconditional process fairness property in the computation space D_1 shown in Fig. 5.1.1 (there is only one alphabet structure $\alpha = \{\{a\},\{b\}\}$ allowed here) is $\Phi_u^{proc}(\alpha) = \{[(ab)^\omega]\}$, which is equal to $Max(\Theta_\iota^\infty)$. Note that $\Phi = \{[a^\omega]\}$ is not a process fairness notion because it is not suffix-closed, $\Phi = \uparrow\{[a^\omega]\}$ is not a process fairness notion because it is not the smallest progress property containing $\{[(ab)^\omega]\}$.
There are four distinct alphabet structures in the computation space D_2 shown in Fig.

5.1.2:

$\alpha_1 = \{\{a,b\}\}$

$\alpha_2 = \{\{a,b\}, \{a\}\}$

$\alpha_3 = \{\{a,b\}, \{b\}\}$

$\alpha_4 = \{\{a,b\}, \{a\}, \{b\}\}$

The minimal alphabet structure here is α_1 and the maximal one is α_4. $\Phi_u{}^{proc}(\alpha_1)$ includes all infinite computations. $\Phi_u{}^{proc}(\alpha_2)$ includes all infinite computations that contain an infinite number of occurrences of a's, while $\Phi_u{}^{proc}(\alpha_3)$ includes all infinite computations that contain an infinite number of occurrences of b's. $\Phi_u{}^{proc}(\alpha_4)$ contains only those infinite computations in which both a and b appear infinitely often. The lattice of process fairness properties over the given alphabet is shown in Fig. 5.2.1.

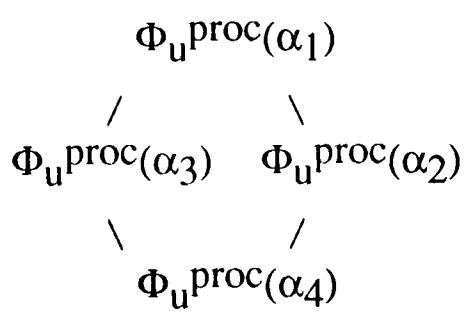


Fig. 5.2.1. Lattice of unconditional process fairness properties (A={a,b}, $\iota = \varnothing$).

Unconditional Event Fairness

Let us again consider a class of asynchronous transition systems with non-terminating processes. We assume that action labels in A correspond to possible events in the system. For such systems, it may be concluded that if an event has been taken finitely often in an infinite admissible computation, then it must have been unfairly delayed. Unconditional event fairness may thus be defined as follows.

Let $C = (A, \iota)$ be a concurrent alphabet, $\alpha \subseteq \wp(A)$ be an alphabet structure over C. We define $\Xi_u{}^{ev} \subseteq \Theta_\iota{}^\omega$ by:

$$\Xi_u{}^{ev} = \{\sigma \in \Theta_\iota{}^\omega \mid \forall\, a \in A: |\sigma|_a = \omega\}$$

where $|\sigma|_a$ denotes the number of occurrences of the symbol a in σ which can be formally defined by $|x/\{a\}|$ for some $x \in \sigma$. A property Φ_u^{ev} is the *unconditional event fairness property* if, and only if, it is the smallest fairness property containing $\uparrow\Xi_u^{ev}$.

The above states that unconditional event fairness property admits all finite computations together with all infinite computations which contain an infinite number of occurrences of actions from the set of actions A. Note that $\cup\{\sigma \in \Phi_u^{ev}\}$ is a *fairmerge* [Par80] of languages $\{a\}^\infty$, for $a \in A$.

Unconditional event fairness exactly coincides with unconditional process fairness $\Phi_u^{proc}(\alpha^{max})$ with respect to the maximal alphabet structure. The following statement is a simple conclusion of the results concerning unconditional process fairness.

Proposition 5.2.7.

(i) $\Phi_u^{ev} = \Phi_u^{proc}(\alpha^{max})$

(ii) $\Phi_u^{ev} \subseteq \Phi_u^{proc}(\alpha)$ for any alphabet structure α.

Proof.

(i) Follows from the fact that α^{max} contains all singleton sets $\{a\}$ for $a \in A$.

(ii) Follows from the fact that α^{max} is a refinement of any other alphabet structure α.

□

5.3. Fairness and Asynchronous Transition Systems

The unconditional fairness properties that have been defined so far were *uniform* fairness properties, that is independent of the asynchronous transition system over the given concurrent alphabet. We now relativize fairness properties with respect to an ATS, thus making it possible to distinguish between a process that does not proceed because it has terminated and a process that is waiting to synchronise.

Relativizing Properties

We relativize properties with respect to a given prefix-closed trace language $T \subseteq \Theta_I^\infty$.

Ψ is a *property* in T if, and only if, $\Psi \subseteq T$. A property $\Psi \subseteq T$ is a *safety property* in T if, and only if, $\Psi = \downarrow\Psi$. A property $\Psi \subseteq T$ is a *progress property* in T if, and only if, $\Psi = \uparrow\Psi$. A progress property $\Psi \subseteq T^{inf}$ is a *fairness property* if, and only if, $T^{fin} \subseteq \downarrow fin\Psi$ whenever Ψ is non-empty.

Let $S = (Q, A, \rightarrow, \iota)$ be an asynchronous transition system, $\Sigma = (S,q)$ be a rooted ATS. Since the set of all computations $T^\infty(\Sigma)$ of a rooted asynchronous transition system is prefix-closed (Proposition 3.6.2), we can relativize properties with respect to Σ.

Fairness and Strength Predicates

We now show how to incorporate strength predicates into our formalism.

Let $C = (A, \iota)$ be a concurrent alphabet, $\alpha \subseteq \wp(A)$ be an alphabet structure over C. Let $S = (Q, A, \rightarrow, \iota)$ be an asynchronous transition system, $\Sigma = (S,q)$ be a rooted ATS. A *strength predicate* ρ is a Boolean function $\rho: T^\omega(\Sigma) \rightarrow B$ such that:

$$T^*(\Sigma) \subseteq \downarrow^{fin}(X) \qquad \text{where } X = \{\sigma \in T^\omega(\Sigma) \mid \rho(\sigma)\}.$$

The class of fairness properties over a given concurrent alphabet with respect to a given strength predicate ρ is now defined as follows. Let $\Xi_\rho(\Sigma) \subseteq T^\omega(\Sigma)$ be given by:

$$\Xi_\rho(\Sigma) = \{\sigma \in T^\omega(\Sigma) \mid \rho(\sigma)\}.$$

A property $\Phi_\rho(\Sigma)$ is a *fairness property with respect to strength predicate* ρ if, and only if, it is the smallest fairness property containing $\uparrow\Xi_\rho(\Sigma)$.

Note that strength predicates are not necessarily determined by alphabet structures.

The following observation follows directly from the definition of a fairness property with respect to a strength predicate.

Observation 5.3.1. (Strength Hierarchy of Fairness Properties)
Let $C = (A, \iota)$ be a concurrent alphabet, $S = (Q, A, \rightarrow, \iota)$ be an asynchronous transition system, $\Sigma = (S,q)$ be a rooted ATS. Let $\rho, \pi: T^{\omega}(\Sigma) \rightarrow B$ be strength predicates. Then the following holds:

If $\rho \Rightarrow \pi$ then $\Phi_{\rho}(\Sigma) \subseteq \Phi_{\pi}(\Sigma)$.

Process Fairness and Strength Predicates

Let $C = (A, \iota)$ be a concurrent alphabet, $\alpha \subseteq \wp(A)$ be an alphabet structure over C. Let $S = (Q, A, \rightarrow, \iota)$ be an asynchronous transition system, $\Sigma = (S,q)$ be a rooted ATS. It is easy to observe that unconditional process fairness properties are determined by the following class of strength predicates $u(\alpha)$ relative to alphabet structures:

$$u(\alpha)(\sigma) = \{\sigma \in \Theta_{\iota}{}^{\omega} \mid \forall\, \alpha_i \in \alpha:\ |p_i(\sigma)| = \omega\}.$$

We define classes of strength predicates that determine weak and strong process fairness:

$w(\alpha)$: $T^{\omega}(\Sigma) \rightarrow B$ (weak process fairness wrt α)

$s(\alpha)$: $T^{\omega}(\Sigma) \rightarrow B$ (strong process fairness wrt α)

as follows:

$$w(\alpha)(\sigma) = (\forall\, \alpha_i \in \alpha, \forall\, \tau \in \mathrm{Pref}^{\mathrm{fin}}(\sigma):$$
$$\alpha_i \text{ continuously enabled in } \sigma/\tau \Rightarrow \alpha_i \text{ taken in } \sigma/\tau)$$

$$s(\alpha)(\sigma) = (\forall\, \alpha_i \in \alpha, \forall\, \tau \in \mathrm{Pref}^{\mathrm{fin}}(\sigma):$$
$$\alpha_i \text{ infinitely often enabled in } \sigma/\tau \Rightarrow \alpha_i \text{ taken in } \sigma/\tau)$$

The precise meaning of the process α_i being *continuously enabled, infinitely often enabled* and *taken* can be found in Chapter 4. Weak process fairness property with respect to α is now defined as the fairness property $\Phi_{w(\alpha)}(\Sigma)$ with respect to the strength predicate $w(\alpha)$. Likewise, strong process fairness property with respect to α is defined as the fairness property $\Phi_{s(\alpha)}(\Sigma)$ with respect to strength predicate $s(\alpha)$.

It is not difficult to convince oneself of the following.

Proposition 5.3.2.
Let C = (A, ι) be a concurrent alphabet, S = (Q, A, →, ι) be an asynchronous transition system, Σ =(S,q) be a rooted ATS. Let $\rho(\alpha)$: $T^{\omega}(\Sigma) \rightarrow B$ be a class of strength predicates such that for all alphabet structures α, α', if α' is a refinement of α then $\rho(\alpha') \Rightarrow \rho(\alpha)$. Then the following holds:

If α' is a refinement of α then $\Phi_{\rho(\alpha')}(\Sigma) \subseteq \Phi_{\rho(\alpha)}(\Sigma)$.

Proof. Clear.

❑

The following is a consequence of the above.

Proposition 5.3.3.
Let C = (A, ι) be a concurrent alphabet, S = (Q, A, →, ι) be an asynchronous transition system, Σ =(S,q) be a rooted ATS. Let α be an alphabet structure over C.

(i) $\Phi_{w(\alpha)}(\Sigma)$ forms a lattice with inclusion ordering.

(ii) $\Phi_{s(\alpha)}(\Sigma)$ forms a lattice with inclusion ordering.

Proof.
Follows from Proposition 5.3.2 and the fact that for every alphabet structure α:

$s(\alpha) \Rightarrow w(\alpha)$.

❑

Example. The hierarchy of process fairness properties for alphabet structures α, α' such that α' is a refinement of α is shown in Fig. 5.3.1. Note that strength predicates other than $w(\alpha)$ and $s(\alpha)$ may also be included.

$$
\begin{array}{ccc}
\Phi_{w(\alpha')}(\Sigma) & \text{———} & \Phi_{w(\alpha)}(\Sigma) \\
| & & | \\
\Phi_{s(\alpha')}(\Sigma) & \text{———} & \Phi_{s(\alpha)}(\Sigma) \\
| & & | \\
\Phi_{u(\alpha')}(\Sigma) & \text{———} & \Phi_{u(\alpha)}(\Sigma)
\end{array}
$$

Fig. 5.3.1. The hierarchy of process fairness properties for alphabet structures $\alpha \leq \alpha'$.

Other Fairness Properties

A variety of fairness properties different from the above process fairness are expressible within our framework.

Equifairness

As an example, we can show that equifairness (see Chapter 2 for definition and overview) is expressible. Let $C = (A, \iota)$ be a concurrent alphabet, $S = (Q, A, \rightarrow, \iota)$ be an asynchronous transition system, $\Sigma = (S,q)$ be a rooted ATS. We define the strength predicate *eqf*: $T^{\omega}(\Sigma) \rightarrow B$ as follows:

$$eqf(\sigma) = (\forall\, D \subseteq A, \forall\, \tau \in \mathrm{Pref}^{\mathrm{fin}}(\sigma):$$
$$D \textit{ jointly enabled inf. often in } \sigma/\tau \Rightarrow D \textit{ taken equally often in } \sigma/\tau).$$

where $D \subseteq A$ is *jointly enabled inf. often in* $\gamma \in T^{\infty}(\Sigma)$ holds iff:

$$\mathrm{card}\{\tau \in \mathrm{Pref}^{\mathrm{fin}}(\gamma) \mid \forall\, a \in D: |x/\{a\}| = |x/\{b\}| \text{ for some } x \in \tau\} = \omega.$$

and D is *taken equally often in* $\gamma \in T^{\infty}(\Sigma)$ is given by:

$$\mathrm{card}\{\tau \in \mathrm{Pref}^{\mathrm{fin}}(\gamma) \mid \forall\, a,b \in D: |x/\{a\}| = |x/\{b\}| \text{ for some } x \in \tau\} = \omega.$$

A property $\Phi_{eqf}(\Sigma)$ is an *equifairness property* if, and only if, it is the fairness property with respect to the strength predicate *eqf* as defined above.

State Fairness

We also show how *fair reachability of states* in its strong form (see Chapter 2 for definition) could be expressed. Let $C = (A, \iota)$ be a concurrent alphabet, $S = (Q, A, \rightarrow, \iota)$ be an asynchronous transition system, $\Sigma = (S,q_0)$ be a rooted ATS. We define the strength predicate *st*: $T^{\omega}(\Sigma) \rightarrow B$ as follows:

$$st(\sigma) = (\forall\, q \in Q, \forall\, \tau \in \mathrm{Pref}^{\mathrm{fin}}(\sigma):$$
$$q \textit{ reachable inf. often in } \sigma/\tau \Rightarrow q \textit{ taken inf. often in } \sigma/\tau)$$

where $q \in Q$ *reachable inf. often in* $\gamma \in T^{\infty}(\Sigma)$ is equivalent to:

$$\text{card}\{\tau \in \text{Pref}^{\text{fin}}(\gamma) \mid \exists\, a \in A: q_0 \rightarrow^{\tau[a]} q\} = \omega$$

and q *taken inf. often in* γ is equivalent to:

$$\text{card}\{\tau \in \text{Pref}^{\text{fin}}(\gamma) \mid \exists\, a \in A: q_0 \rightarrow^{\tau} q\} = \omega.$$

A property $\Phi_{st}(\Sigma)$ is a *strong state fairness property* if, and only if, it is the fairness property with respect to the strength predicate *st* as defined above.

6

Applications of Theory to Condition/Event Nets

Asynchronous transition systems are a natural extension of sequential labelled transition systems with the notion of independency relation. Since many models for concurrency can be viewed as a state-transition system, they obviously can be given semantics in terms of a labelled transition system, which corresponds to a *sequential* asynchronous transition system. Models included in this class are process algebras like CCS [Mil80] with standard transition system semantics and TCSP [BHR84].

A more relevant question is whether existing models for concurrency determine *non-sequential* asynchronous transition systems. The answer to this is, in many cases, positive, as it turns out that the independency relation is a *syntactical* notion, rather than a *behavioural* notion of concurrency relation distinguished in event structures [Win86]. Thus, it is possible to determine independency from the terms of the process algebra; likewise, it is possible to determine independency from the structure of a net.

In this chapter, we show, as an example of applications of the theory developed in the thesis, that Condition/Event nets [RoT86] determine asynchronous transition systems. We also provide Condition/Event nets with trace semantics. Finally, we discuss the classes of properties of Condition/Event nets in this setting. The results concerning trace semantics are not new; we have included them for completeness' sake mainly to demonstrate the applications of the mathematical space of behavioural properties in the computation space of traces.

6.1. Condition/Event Nets

The theory of Petri nets originated from Petri in the early 1960's as an attempt to mathematically formalize phenomena present in information systems such as information flow, conflict, concurrency and synchronisation. The notion central to net theory is that of causality, and the major attraction is a syntactical distinction between concurrency and non-determinism. The latter feature puts net theory in a class of non-interleaving models for concurrency. Net theory has been greatly enhanced in the past. It incorporates a variety of classes of nets, one of which is the class of Condition/Event nets.

We first introduce basic definitions relating to Petri nets. These are due to [Rei85] [RoT86]. Below is a definition of an abstract Petri net, which describes the structure of the system, rather than its behaviour.

Definition.

A (finite) Petri net is a triple $N = (S, T; F)$, where S is a finite set of *places*, T is a finite set of *transitions*, and F is a set of *arcs* (or *flow relation*), such that $S \cap T = \emptyset$, and $F \subseteq (S \times T) \cup (T \times S)$. We write

$$\bullet x = F^{-1}(x),\ x\bullet = F(x)$$ (the *preset* and *postset* correspondingly)

for $x \in S \cup T$.

The net N is called *pure* iff it does not contain any self-loops, that is pairs

$(s, t) \in S \times T$ such that $(s, t) \in F$ and $(t, s) \in F$.

In order to describe the behaviour of the system given as a Petri net, one needs to add firing rules. It is assumed that each place can hold a number of tokens. A transition fires if it has at least one token in its input places, which results in tokens to be added to the output places of the transition.

Condition/Event nets are a subclass of Petri nets which are pure and are restricted to having at most one token in each place. This simplifies the firing rule. In a Condition/Event net, we refer to the set of places as the set B of *conditions*, the set of transitions as the set E of *events*.

Definition.

Let N = (B, E; F) be a net.

(i) A subset c of B is called a *case*.

(ii) Let $e \in E$ and $c \subseteq B$. e is c-*enabled* iff $\bullet e \subseteq c \wedge e\bullet \cap c = \varnothing$.

(iii) Let $e \in E$, let $c \subseteq B$ and let e be c-enabled. $c' = (c \setminus \bullet e) \cup e\bullet$ results from the *occurrence of* e *in the case* c and we write: $c [e > c'$.
The relation $\_ [\_ > \_$ is called the *firing relation.*

(iv) Let $c_{in} \subseteq B$. The pair (N, c_{in}) is called a *Condition/Event system.* c_{in} is the *initial case.*

Examples of Condition/Event nets are shown in Fig.6.1.1 and Fig.6.1.2.

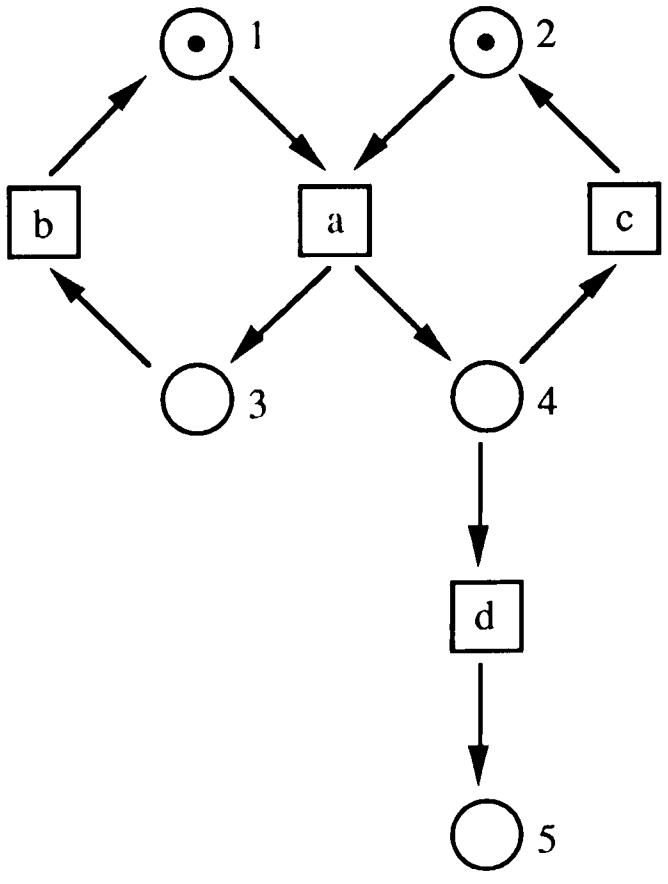


Fig. 6.1.1. An example of a Condition/Event net.

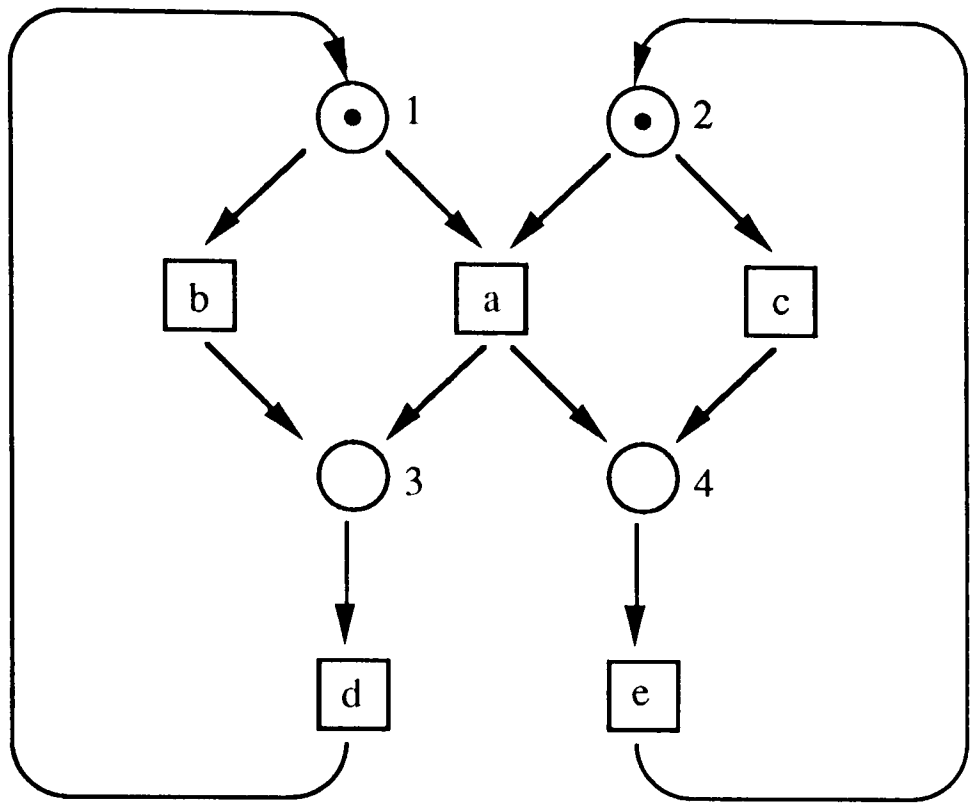


Fig. 6.1.2. An example of a Condition/Event net.

Example. Formally, the net shown in Fig. 6.1.1 is a triple N = (B, E; F) given by:

B = {1,2,3,4,5}

E = {a,b,c,d}

F = {(1,a), (2,a), (a,3), (a,4), (3,b), (b,1), (4,c), (c,2), (4,d), (d,5)}

The initial case of N is {1,2}. a is {1,2}-enabled, but d is not. Also, •a = {1,2}, a• = {3,4}, •1 = {b}, 1• = {a}. The Condition/Event system here is (N, {1,2}), where {1,2} is the initial case.

6.2. Asynchronous Semantics for Condition/Event Nets

ATS Semantics

We shall now define an asynchronous transition system that arises from a given Condition/Event net. The set of states will be defined as the set $\wp(B)$ of all the cases, the set of action labels as the set E of events of the net, the transition relation will be the firing relation $\_ [\_ > \_$, and, finally, two actions will be independent if, and only if, they are potentially concurrent, that is the immediate neighbourhoods (the union of the preset and the postset) of the corresponding transitions are disjoint.

Definition. Given a Condition/Event net $N = (B, E; F)$, let us define $S_N = (Q, A, \rightarrow, \iota)$ by:

(i) $Q = \wp(B)$

(ii) $A = E$

(iii) $\rightarrow = Q \times A \times Q \cap \_ [\_ > \_$

(iv) $e_1 \iota e_2 \Leftrightarrow (\bullet e_1 \cup e_1 \bullet) \cap (\bullet e_2 \cup e_2 \bullet) = \emptyset$, for $e_1, e_2 \in E$.

Proposition 6.2.1. S_N is an unambiguous asynchronous transition system.

Proof. Clearly, $(Q, A, \rightarrow)$ is a transition system, ι is irreflexive and symmetric.

We shall now prove condition (i) of definition of an asynchronous transition system. Let us assume that $e_1 \iota e_2$, for some $e_1, e_2 \in E$, and there exists a firing sequence:

$$c [e_1 > c' [e_2 > c'',$$

for $c, c', c'' \in \wp(B)$. We shall show that there exists $c''' \in \wp(B)$ such that

$$c [e_2 > c''' [e_1 > c''.$$

Note that $e_1 \iota e_2$ is equivalent to the conjunction of the following conditions:

(a) $\bullet e_1 \cap \bullet e_2 = \emptyset$

(b) $e_1 \bullet \cap e_2 \bullet = \emptyset$

(c) $\bullet e_1 \cap e_2 \bullet = \emptyset$

(d) $e_1 \bullet \cap \bullet e_2 = \emptyset$

We shall first prove that e_2 is c-enabled.

	$\bullet e_2 \subseteq c'$	because e_2 is c'-enabled
$\Rightarrow$	$\bullet e_2 \subseteq (c \setminus \bullet e_1) \cup e_1 \bullet$	from def. $c' = (c \setminus \bullet e_1) \cup e_1 \bullet$
$\Rightarrow$	$\bullet e_2 \subseteq c$	from (c) and (d)

	$e_2 \bullet \cap c' = \emptyset$	because e_2 is c'-enabled
$\Rightarrow$	$e_2 \bullet \cap ((c \setminus \bullet e_1) \cup e_1 \bullet) = \emptyset$	
$\Rightarrow$	$(e_2 \bullet \cap (c \setminus \bullet e_1)) \cup (e_2 \bullet \cap e_1 \bullet) = \emptyset$	
$\Rightarrow$	$e_2 \bullet \cap (c \setminus \bullet e_1) = \emptyset$	from (b)
$\Rightarrow$	$(e_2 \bullet \cap c) \setminus (e_2 \bullet \cap \bullet e_1) = \emptyset$	from $[x \cap (y \setminus z) = [(x \cap y) \setminus (x \cap z)]$
$\Rightarrow$	$e_2 \bullet \cap c = \emptyset$	from (c)

Let $c''' = (c \setminus \bullet e_2) \cup e_2 \bullet$. We shall now show that e_1 is c'''-enabled.

	$\bullet e_1 \subseteq c$	because e_1 is c-enabled
$\Rightarrow$	$\bullet e_1 \subseteq c \setminus \bullet e_2$	from (a), $[x \subseteq y, x \cap z = \emptyset \Rightarrow x \subseteq y \setminus z]$
$\Rightarrow$	$\bullet e_1 \subseteq c'''$	

	$e_1 \bullet \cap c'''$	
=	$e_1 \bullet \cap ((c \setminus \bullet e_2) \cup e_2 \bullet)$	distributivity of $\cap$
=	$(e_1 \bullet \cap (c \setminus \bullet e_2)) \cup (e_1 \bullet \cap e_2 \bullet)$	from definition of c'''
=	$(e_1 \bullet \cap (c \setminus \bullet e_2))$	from (b)
=	$\emptyset$	because $e_1 \bullet \cap c = \emptyset$

Finally, we show that $c''' [e_1 > c''$.

	$(c''' \setminus \bullet e_1) \cup e_1 \bullet$	
=	$[((c \setminus \bullet e_2) \cup e_2 \bullet) \setminus \bullet e_1] \cup e_1 \bullet$	
=	$[((c \setminus \bullet e_2) \setminus \bullet e_1) \cup (e_2 \bullet \setminus \bullet e_1)] \cup e_1 \bullet$	from $[(x \cup y) \setminus z = (x \setminus z) \cup (y \setminus z)]$
=	$((c \setminus \bullet e_2) \setminus \bullet e_1) \cup e_2 \bullet \cup e_1 \bullet$	because (c) $\Rightarrow (e_2 \bullet \setminus \bullet e_1 = e_2 \bullet)$

$$
\begin{aligned}
&= ((c \setminus \bullet e_1) \setminus \bullet e_2) \cup e_2\bullet \cup e_1\bullet && \text{from } [(x\setminus y)\setminus z = (x\setminus z)\setminus y] \\
&= ((c \setminus \bullet e_1) \setminus \bullet e_2) \cup (e_1\bullet \setminus \bullet e_2) \cup e_2\bullet && \text{from (d)} \\
&= [((c \setminus \bullet e_1) \cup e_1\bullet) \setminus \bullet e_2] \cup e_2\bullet && \text{from } [(x\setminus z)\cup(z\setminus y) = (x\cup z)\setminus y] \\
&= (c' \setminus \bullet e_2) \cup e_2\bullet \\
&= c''' \\
&\text{----------}
\end{aligned}
$$

A similar argument can be used to show condition (ii) of the definition of ATS. Conditions (c), (d) are not required for this case. Hence, S_N is forward stable.

Finally, it should be observed that if $e \in E$, $c \subseteq B$, e is c-*enabled* and $c [e > c'$, then c' is uniquely determined, and thus $\rightarrow^e$ is a (partial) function. Therefore, S_N is unambiguous. It is also finite, as the net is assumed to be finite, and hence there are no more than card(2^B) cases.

❑

Let N = (B, E; F) be a Condition/Event net, $c_{in} \subseteq B$ be the initial case of the net, and S_N = (Q, A, $\rightarrow$, ι) be the asynchronous transition system determined by the net. Then Σ =(S_N, c_{in}) is a rooted asynchronous transition system determined by the Condition/Event system (N, c_{in}).

Example. Fig. 6.2.1 and Fig.6.2.2 show the asynchronous transition systems determined by the Condition/Event nets included in Fig. 6.1.1 and 6.1.2 respectively. (All cases that are not reachable from the initial case have been omitted.) The ATS shown in Fig. 6.2.1 is non-sequential, non-determinate and confusion-free (b ι c, b ι d). The ATS shown in Fig. 6.2.2 is non-sequential, non-determinate and exhibits both symmetric and asymmetric confusion (b ι c, b ι e, d ι c, d ι e).

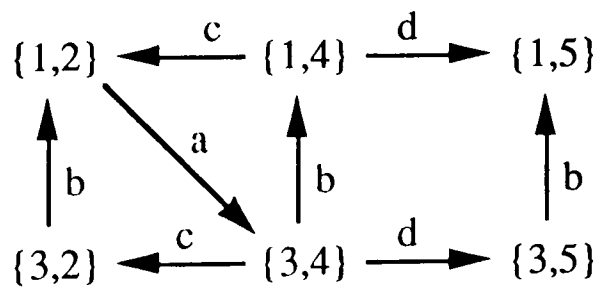


Fig. 6.2.1. The ATS determined by the net shown in Fig. 6.1.1.

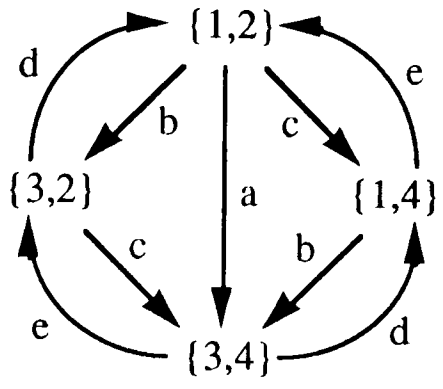


Fig. 6.2.2. The ATS determined by the net shown in Fig. 6.1.2.

Interleaving Semantics

Let N = (B, E; F) be a Condition/Event net, $c_{in} \subseteq B$ be the initial case of the net, $\Sigma_N = (S_N, c_{in})$ with $S_N = (Q, A, \rightarrow, \iota)$ be a rooted asynchronous transition system determined by the system (N, c_{in}). We define the interleaving semantics of the net N as the set of derivations of the asynchronous transition system S_N:

$D^\infty(S_N)$.

The set of derivations of the Condition/Event system (N, c_{in}) is defined as:

$D^\infty(\Sigma_N)$.

The set of derivations of the net corresponds exactly to firing sequences. It follows from Proposition 3.4.1 that $D^\infty(\Sigma_N)$ is a prefix-closed and closed infinitary string language ordered by prefix ordering. The prefix ordering ignores causality, and thus does not distinguish the ordering of events obtained due to concurrency from the ordering arising due to conflict resolution. Also, some infinite derivations may only represent local, therefore subjective, view of the system behaviour.

Example. The set of derivations determined by the Condition/Event system shown in Fig. 6.1.1 is given by the prefix closure of:
$((abc)^*(acb)^*)^* \cup ((abc)^*(acb)^*)^\omega \cup ((abc)^*(acb)^*)^*bd \cup ((abc)^*(acb)^*)^*db$.
The set of derivations determined by the system in Fig. 6.1.2 is the prefix closure of:

$((bd)^*(ade)^*(ce)^*)^* \cup ((bd)^*(ade)^*(ce)^*)^\omega$.

Note that the latter admits $(bd)^\omega$, although c is permanently enabled.

Trace Semantics

Causality can be incorporated into the behaviour of nets by means of the independency relation, which gives rise to the additional structure over the set of all derivations.

Let $N = (B, E; F)$ be a Condition/Event net, $c_{in} \subseteq B$ be the initial case of the net, $\Sigma_N = (S_N, c_{in})$ with $S_N = (Q, A, \rightarrow, \iota)$ be a rooted asynchronous transition system determined by the system (N, c_{in}). We define the trace semantics of the net N as the set of traces of the asynchronous transition system S_N:

$T^\infty(S_N)$.

The set of traces determined by the system (N, c_{in}) is defined as:

$T^\infty(\Sigma_N)$.

The *admissible trace behaviour* of the net is $Adm(\Sigma_N) = T^*(\Sigma_N) \cup Max(T^\infty(\Sigma_N))$. The *admissible sequential behaviour* (i.e. interleaving semantics) is defined as $\cup\{\sigma \in Adm(\Sigma_N)\}$.

It follows from Proposition 3.6.4 that $T^\infty(\Sigma_N)$ is a prefix-closed and closed infinitary trace language ordered by trace prefix ordering. We also have (Proposition 3.6.1) that every derivation contained in a trace is a firing sequence of the net, and that every trace uniquely determines a case.

Example. The admissible trace behaviour of the Condition/Event system shown in Fig. 6.1.1 is given by the trace prefix closure of:

$[(abc)^*] \cup [(abc)^\omega] \cup |(abc)^{**}bd|$.

The admissible trace behaviour of the system in Fig. 6.1.2 is the finitary trace prefix closure of:

$[(bd(bd)^*(ade)^*ce(ce)^*)^*| \cup |(bd(bd)^*(ade)^*ce(ce)^*)^\omega]$.

Note that this excludes $(bd)^\omega$.

Process Structures

A variety of alphabet structures are possible over a given Condition/Event net. Let N = (B, E; F) be a Condition/Event net, $c_{in} \subseteq B$ be the initial case of the net, $\Sigma_N = (S_N, c_{in})$ with $S_N = (Q, A, \rightarrow, \iota)$ be a rooted asynchronous transition system determined by the system (N, c_{in}). Let α be an alphabet structure over (A, ι). Then for each $\alpha_i \in \alpha$ we have:

$$a, b \in \alpha_i \Leftrightarrow (\bullet a \cup a\bullet) \cap (\bullet b \cup b\bullet) \neq \varnothing$$

or, in other words, two events belong to the same alphabet if, and only if, they share a condition. Thus, we have decomposed the net into *sequential* connected subnets. Let us define for each $\alpha_i \in \alpha$ the subnet $N_i = (B_i, E_i; F_i)$ as follows:

$$B_i = \{\bullet a \cup a\bullet \mid a \in \alpha_i\}$$
$$E_i = \alpha_i$$
$$F_i = F \cap ((B_i \times E_i) \cup (E_i \times B_i))$$

Clearly, if $\alpha_i \subseteq \alpha_j$ then $B_i \subseteq B_j$, $E_i \subseteq E_j$ and $B_i \subseteq B_j$, and thus $N_i \subseteq N_j$. A consequence of this is that, whenever α is a refinement of α', we have, for any $\alpha_i' \in \alpha'$, there exists $\alpha_j \in \alpha$ such that $N_i' \subseteq N_j$. Thus, the minimal alphabet structure α^{min} determines a decomposition of the given net into maximal connected sequential subnets. The maximal alphabet structure α^{max}, on the other hand, contains a decomposition into subnets consisting of exactly one event together with all surrounding conditions.

We define a *process structure* $\Pi_N = (\alpha,\pi)$ over a Condition/Event net N as the process structure over the ATS S_N determined by the net (Σ_N respectively). A process $\alpha_i \in \alpha$ is *enabled* in the case $c \subseteq B$ iff there exists $a \in \alpha_i$ such that $\bullet a \subseteq c$.

It follows from Proposition 3.3.1 that decompositions of a given net with respect to alphabet structures form a lattice with inclusion ordering. Refining alphabet structures corresponds to the increase in the granularity of decompositions.

Example. Examples of alphabet structures over the net shown in Fig. 6.1.1 are:

$\alpha^{min} = \{\{a,b\},\{a,c,d\}\}$

$\alpha^{max} = \{\{a,b\},\{a,c,d\},\{a,c\},\{a,d\},\{c,d\},\{a\},\{b\},\{c\},\{d\}\}$.

The following are examples of alphabet structures over the net shown in Fig. 6.1.2:

$\alpha^{min} = \{\{a,b,d\},\{a,c,e\}\}$

$\alpha = \{\{a,b,d\},\{a,c,e\},\{a,b\},\{a,d\}\}$.

Vector Semantics

Let N = (B, E; F) be a Condition/Event net, $c_{in} \subseteq B$ be the initial case of the net, $\Sigma_N = (S_N, c_{in})$ with $S_N = (Q, A, \rightarrow, \iota)$ be a rooted asynchronous transition system determined by the system (N, c_{in}). Let α be an alphabet structure over (A, ι). We define vector semantics of the net N as the set of vectors determined by the asynchronous transition system S_N:

$$V^{\infty}(S_N).$$

The set of vectors determined by the system (N, c_{in}) is defined as:

$$V^{\infty}(\Sigma_N).$$

Vector semantics allows us to localize the behaviour of the net with respect to a given process structure. Let $\Pi_N = (\alpha,\pi)$ be a process structure, then for each $\alpha_i \in \alpha$, for each $\underline{w} \in V^{\infty}(S_N)$ we have w_i represents the (sequential) behaviour of the subnet N_i.

6.3. Safety, Progress and Fairness for Condition/Event Nets

Let N = (B, E; F) be a Condition/Event net, $c_{in} \subseteq B$ be the initial case of the net, $\Sigma_N = (S_N, c_{in})$ with $S_N = (Q, A, \rightarrow, \iota)$ be a rooted asynchronous transition system determined by the system (N, c_{in}). Let us consider the class of properties Ψ in the computation space Θ_{ι}^{∞} over the concurrent alphabet (A, ι). The net N is said to *satisfy* property Ψ if, and only if, Σ_N satisfies Ψ, that is, $Adm(\Sigma_N) \subseteq \downarrow^{fin}\Psi$.

Safety Properties

Let (A, ι) be a concurrent alphabet. *Mutual exclusion*, a property stating that no two processes simultaneously possess a resource, is an example of a safety property. Suppose α is an alphabet structure over (A, ι), and let $\alpha_i, \alpha_j \in \alpha$ be processes. Let $x_i \in \alpha_i$, $x_j \in$

α_j represent the events of the resource being granted to the respective processes, $y_i \in \alpha_i$, $y_j \in \alpha_j$ correspond to the resource being released (note that x_i, x_j, y_i, y_j must be pairwise dependent because they refer to the same resource). Then mutual exclusion of processes α_i, α_j is the property $\Psi_{i,j}^{mex}$ defined as follows:

$$\Psi_{i,j}^{mex} = \{\sigma \in \Theta_\iota^\infty \mid \mathit{mutex}(\sigma,i,j) \wedge \mathit{mutex}(\sigma,j,i)\}$$

where $\mathit{mutex}(\sigma,k,m)$ is a predicate that ensures that x_k and y_k are excluded in σ between two consecutive occurrences of x_m and y_m (note that events other than x_k and y_k should not be excluded). Formally, $\mathit{mutex}(\sigma,k,m)$ holds if, and only if:

$$(\sigma = \sigma'[x_m]\sigma''[y_m]\sigma''' \wedge p_m(\sigma'') = \varepsilon) \Rightarrow ((\mathit{occ}(\sigma'',x_k) = 0) \wedge (\mathit{occ}(\sigma'',y_k) = 0))$$

where $\mathit{occ}(\gamma,a)$ denotes the number of occurrences of the symbol a in the trace γ.

It is easy to see that $\Psi_{i,j}^{mex}$ is prefix-closed and closed, but it is not suffix-closed (there are extensions of σ which are not in the property).

Example. Let us consider the Condition/Event net shown in Fig. 4.2.1 that implements mutual exclusion. Assuming the minimal process structure α^{min} with $\alpha_1 = \{a,b,c\}$, $\alpha_2 = \{d,e,f\}$, $\alpha_3 = \{a,b,d,e\}$, we have a and d correspond to the requests being granted while b and e represent the release. It is easy to see that the net satisfies $\Psi_{1,2}^{mex}$ as the admissible trace behaviour is $\downarrow^{fin}[((abc)^*(def)^*)^\omega]$.

Another example of an abstract safety property would be one that states that *any resource released must have been previously granted.* Let us again consider an alphabet structure α over (A, ι), and let $\alpha_i \in \alpha$ be a process such that $x_i \in \alpha_i$ is the event of the resource being granted, and $y_i \in \alpha_i$ represents release. We can now define the above property as the property Ψ_i^{rel} of the process α_i as follows:

$$\Psi_i^{rel} = \{\sigma \in \Theta_\iota^\infty \mid \mathit{rel}(\sigma,i)\}$$

where $\mathit{rel}(\sigma,i)$ is a predicate that is true if, and only if:

$$(\sigma = \sigma'[y_i]\sigma'') \Rightarrow ((\sigma' = \gamma'[x_i]\gamma'') \wedge (\mathit{occ}(\gamma'',x_i) = 0) \wedge (\mathit{occ}(\gamma'',x_i) = \mathit{occ}(\gamma'',y_i))).$$

It is clear that Ψ_i^{rel} is prefix-closed and closed, but not suffix-closed.

Example. Let us again consider the Condition/Event net shown in Fig. 4.2.1 with α^{min} where $\alpha_1 = \{a,b,c\}$, $\alpha_2 = \{d,e,f\}$, $\alpha_3 = \{a,b,d,e\}$. Clearly, the net satisfies $\Psi_1{}^{rel}$ and $\Psi_2{}^{rel}$.

Progress Properties

Let (A, ι) be a concurrent alphabet. An example of a finitary progress property is that of *guaranteed response,* which is usually defined as *every resource that has been requested will eventually be granted.* We simplify the definition of this property to the statement that any resource will *eventually* be granted. Suppose α is an alphabet structure over (A, ι), let $\alpha_i \in \alpha$ be a process, and let $x_i \in \alpha_i$ represent the event of the resource being granted. Then guaranteed response for the process α_i, is the property $\Psi_i{}^{res}$ defined as follows:

$$\Psi_i{}^{res} = \{\sigma \in \Theta_\iota{}^{\infty} \mid occ(\sigma,x_i) \geq 1\}.$$

Note that $\Psi_i{}^{res}$ is not prefix-closed, but it is suffix-closed.

Example. Let us again consider the Condition/Event net shown in Fig. 4.2.1 that implements mutual exclusion between processes $\alpha_1 = \{a,b,c\}$ and $\alpha_2 = \{d,e,f\}$, while $\alpha_3 = \{a,b,d,e\}$ is the scheduler (assuming the minimal process structure α^{min}). Since a and d correspond to the requests for the resource being granted, we may wish to determine if the net satisfies guaranteed response. Unfortunately, the answer is negative, as admissible trace behaviour of this net, namely $\downarrow^{fin}[((abc)^*(def)^*)^\omega]$, is *not* contained in $\downarrow^{fin}\Psi_1{}^{res}$ as it allows $[(def)^\omega]$. The same problem arises with any other alphabet structure. The only solution to the problem seems to be to restrict the set of admissible trace behaviour by incorporating fairness assumptions.

The above-mentioned property of guaranteed response was finitary, as an infinite number of finite traces have this property. We now consider an example of an *infinitary* progress property which is a stronger version of guaranteed response; it states that any resource will be granted *infinitely often*. Suppose α is an alphabet structure over (A, ι), and let $\alpha_i \in \alpha$ be a process, and let $x_i \in \alpha_i$ represent the event of the resource being granted. Then infinitary guaranteed response for the process α_i, is the property $\Psi_i{}^{\omega res}$ defined as follows:

$$\Psi_i{}^{\omega res} = \{\sigma \in \Theta_\iota{}^{\infty} \mid occ(\sigma,x_i) = \omega\}.$$

It is easy to see that $\Psi_i{}^{\omega res} \subseteq \Theta_\iota{}^{\omega}$. Also, $\Psi_i{}^{\omega res}$ is not prefix-closed, but it is suffix-closed.

Example. Obviously, the Condition/Event net shown in Fig. 4.2.1 does not satisfy $\Psi_1{}^{\omega res}$ nor $\Psi_2{}^{\omega res}$.

Fairness Properties

Fairness properties are a subclass of infinitary progress properties. The feature that distinguishes fairness from infinitary progress properties is that every finite trace can be extended to a fair trace. In fact, $\Psi_i{}^{\omega res}$ is a fairness property because $\Theta_\iota{}^* \subseteq \downarrow fin\Psi_i{}^{\omega res}$.

Let us consider the algebra of process fairness properties and their relationship with Condition/Event nets. Let α be an alphabet structure over (A, ι), then, as we observed earlier, α determines a decomposition of the net into sequential connected subnets N_i. The unconditional process fairness property $\Phi_u{}^{proc}(\alpha)$ contains those, and only those, infinite traces in which each $\alpha_i \in \alpha$ is taken infinitely often. This corresponds to some event in each subnet being taken infinitely often; if the subnet contains a conflict, either of the events at conflict may occur. Two extreme cases are α^{min} and α^{max}; the first corresponds to the decomposition into maximal sequential subnets, while the latter decomposes the net into single transitions, thus imposing fair resolution of *every* conflict encountered in the net.

Let N = (B, E; F) be a Condition/Event net, $c_{in} \subseteq B$ be the initial case of the net, $\Sigma_N = (S_N, c_{in})$ with $S_N = (Q, A, \rightarrow, \iota)$ be a rooted asynchronous transition system determined by the system (N, c_{in}). Let α be an alphabet structure over (A, ι). For a given alphabet structure, i.e. a decomposition of the net, the unconditional, strong and weak process fairness properties $\Phi_u{}^{proc}(\alpha)$, $\Phi_S{}^{proc}(\alpha)$ and $\Phi_w{}^{proc}(\alpha)$ form a hierarchy. The class of all unconditional (weak, strong respectively) process properties over a given concurrent alphabet forms a lattice.

Example. Let us consider the net shown in Fig. 6.1.2. The computation $[(bdce)^\omega]$ is process fair wrt $\alpha^{min} = \{\{a,b,d\},\{a,c,e\}\}$, but it is not process fair wrt:

$\alpha = \{\{a,b,d\},\{a,c,e\},\{a,b\},\{a,d\},\{a,c\},\{a,e\},\{a\}\}$.

All distinct process fairness properties for this net are:

$[(bd(bd)^*(ade)^*ce(ce)^*)^\omega]$

$[(bd(bd)^*ade(ade)^*(ce)^*)^\omega]$

$[((bd)^*ade(ade)^*ce(ce)^*)^\omega]$
$[((bd)^*ade(ade)^*(ce)^*)^\omega]$
$[(bd(bd)^*(ade)^*ce(ce)^*)^\omega]$
$[(bd(bd)^*ade(ade)^*ce(ce)^*)^\omega]$.

It should be noted that process fairness properties enforce fairness with respect to groups of *events* that become enabled. It is also possible to introduce state fairness into Condition/Event nets; this notion enforces fairness with respect to reachability of *cases*.

Example. Let us consider the net shown in Fig. 6.1.1. The computation $[(abc)^\omega]$ of this net is process fair with respect to $\alpha^{min} = \{\{a,b\},\{a,c,d\}\}$. However, it is not fair with respect to the case $\{5\}$ which is reachable infinitely often (each computation $[(abc)^*a]$, $[(abc)^*ab]$ enables $\{5\}$), but never taken (because for no finite prefix σ of $[(abc)^\omega]$ do we have $\{1,2\} \rightarrow^\sigma \{5\}$).

Note that the class of fairness properties for Condition/Event nets is not restricted to the properties mentioned so far; fairness properties like *probabilistic* fairness, where each transition is assigned a probability weighting, could easily be expressed.

Summary

We have briefly shown how the theory developed in the previous chapters may be applied to Condition/Event nets. We have strong reasons to believe that some safety and progress properties investigated in formalisms like temporal logic [Pnu86] can be naturally expressed in our formalism.

A number of fairness properties, including process, event and state fairness, could be defined in our formalism. Given a variety of alphabet structures over a concurrent alphabet, it is not clear which process fairness property is adequate for a given Condition/Event net. This seems to suggest that (uninterpreted) Condition/Event nets are a low-level model, where entities that represent concurrent synchronising agents are not clearly defined. In order to guarantee uniqueness of the decomposition, additional information would have to be provided.

7

Conclusion

7.1. Summary of Results

General Observations

Although, on the whole, we are pleased with the outcome of the thesis, we wish more of the tasks set out at the beginning could be completed. It came as an unpleasant surprise that the subject of fairness was in a state of confusion; the difficulty was compounded by the sheer number of publications in the area, often guided by different objectives and implicit assumptions. It seemed important at the time to understand the motivation and the inter-relationships of most existing fairness notions; unfortunately, this proved a formidable task. It also seemed unfair on the subject not to include an overview of prevailing approaches, no matter how irrelevant from the point of view of the object of the thesis. As a result, perhaps too much time has been spent on trying to find out more about fairness notions, in most cases informally defined, and no time was left to carry out a formal comparative study of existing fairness properties using the model as a reference.

We have nevertheless succeeded in presenting a unified approach to defining behavioural properties of concurrent systems which includes safety, progress and fairness properties. Using a non-interleaving semantic model based on the notion of causal independency, it has been shown that it is possible to construct a comprehensive mathematical space of properties, within which fairness notions are a subclass of infinitary progress properties. An algebra of unconditional process fairness properties, shown to form a lattice with inclusion ordering, has also been identified. Every process fairness property in this algebra is determined by an alphabet structure that arises from the concurrent alphabet, namely a set of actions together with the independency relation. Both concurrency fairness, including synchronisation fairness, and fairness of choice are expressible. There exist unique strongest and weakest process fairness properties, which are exactly the intersection and the union of all process fairness properties. The strongest process fairness corresponds to stating that *every action must happen*, while the weakest process fairness is, in general, stronger than the statement *every action may happen*. This is because we distinguish between causally dependent and independent actions; thus, the weakest process fairness

would enforce that an action *must* happen as long as it is *permanently possible* and *independent* of the remaining actions in a computation.

The topological characterization of behavioural properties presented here is based on the Scott topology of finitary progress properties. In this topology, safety properties are the closed sets, infinitary progress properties are the G_δ-sets, and fairness properties are the dense sets. The formalism is not limited to unconditional process fairness only; by means of strength predicates fairness properties may be relativized with respect to a given rooted asynchronous transition system. A formalization of weak and strong process fairness properties has been outlined; fairness properties such as state fairness and equifairness are expressible as well.

We hope that our work together with the bibliography would be of help to anyone interested in fairness.

Advantages of the Presented Approach

The approach presented here relies on the assumption that *concurrency* and *causality* are primitive; both are derivable from a notion of causal independency. Thus, although the observations in the model are essentially sequences of system states, the semantics is richer in the sense that non-deterministic choice is distinguished from concurrency. This means that fairness of choice due to internal non-determinism and fairness of synchronisation conflicts are not confused as it may be the case of the interleaving approach.

It should be noted that, although we have not fully analysed the relationship of the interleaving and trace semantics, one conclusion is quite clear: we can replace the *justice* assumption [Pnu86] with the notion of *maximality* of computations in the domain of traces. We believe that the model presented here is a more adequate reflection of reality as far as the behavioural aspects of asynchronous systems are concerned. No assumption of the existence of a global clock has been made, nor has the behaviour been reduced to a synchronous one (we have, however, assumed atomicity of actions).

It should also be stressed that causal independency is a *syntactical* notion; as such, it describes the *potential* for concurrency, rather than the fact that concurrent actions have been *observed*. It turns out that independency is determined by the syntax of the language; for Condition/Event nets it is the structure of the net, for CCS it is the syntax of a closed

term. Although we have only showed that Condition/Event systems determine asynchronous transition systems, it seems plausible that a similar result can also be derived for many existing models for concurrency, e.g. fair transition systems [Pnu86] and CCS [Mil80], thus providing these models with asynchronous semantics. This means that the results concerning fairness and progress are *not* model-specific, but could be transferred to other settings without major problems.

The main contribution of the thesis, however, is a uniform approach to progress and fairness properties. This proves that, despite wide-spread disagreement and confusion on the subject of fairness, it is possible to formulate a comprehensive theoretical basis for the study of fairness. Indeed, when viewing fairness as an issue to do with concurrency in the sense that no concurrent process that becomes possible sufficiently often should be indefinitely delayed, the corresponding fairness properties turn out to form an algebra which is closed under union and intersection.

Related Work

Trace languages have been developed in [Maz77] [Maz84a] [Maz84b] [Bed87] [AaR88], where only finite traces and finitary trace languages have been considered. A notion related to that of a trace is a *dependency graph* [AaR88] [Maz88], which could be infinite.

Asynchronous transition systems were introduced in [Shi85a] [Bed87] [Shi88c] [Kwi88a]. We use a simplified notion of a process structure, which does not distinguish between two processes with the same alphabet. The notion of the projective preorder and equivalence is due to Shields [Shi88c], who also shows that infinitary vector languages form a partial monoid. Vector semantics has also been defined in [Fau87]; infinite behaviours are included, but the approach is model-specific.

Infinitary sequential languages were introduced in [BoN79] [Par80] [Par81], and their relationship with fairness was considered in [Par80] [Par81]. A number of notions of fairness for finite state automata were formulated in [PRW87]. The approach presented here is different as we allow a possibly countable set of states.

A topological characterization of safety and liveness can be found in [AlS85], where liveness is not closed under intersection (fairness is not dealt with). A further difference is that an interleaving semantic model, which gives rise to a different space of behaviours, is

used in [AlS85]. Safety properties and inevitability properties, corresponding to liveness, in a model based on partial order are defined in [MOP88]. Inevitability properties are not closed under intersection.

7.2. Further Developments

Many further developments of the theory are possible. We give a systematic overview of the issues that arise from our research.

Trace Languages

Trace languages constitute a very elegant representation of concurrent behaviours that are a generalization of sequential behaviours. They have received a lot of attention recently [AaR88] [Maz88]; as a result, a number of theoretical problems have been solved. It seems, however, that infinitary trace languages have not yet been fully investigated. The following require investigation: closure properties of infinitary trace languages; existence of automata that recognise infinitary trace languages; ω-regular infinitary trace languages; morphism of infinitary trace languages.

Metrics and Closures of Trace Languages

We have not been able to define a metric over Θ_ι^∞ that corresponds to the closure in terms of adherence. This is unsatisfactory as in the monoid of strings there is a natural metric (ultra-metric) [BoN79] defined by :

$$d(x, y) = 2^{-n}$$

where n is the length of the longest common prefix of strings x and y, which exactly corresponds to the closure of string languages by means of adherence. Unfortunately, a naive attempt to transfer this definition to Θ_ι^∞ using:

$$d'(\sigma, \gamma) = 2^{-n}$$

where n is the longest common prefix (with respect to trace prefix ordering) fails. The following is a counter-example:

$A = \{a,b,c,d\}$, $a \iota b$, $b \iota c$. Then $abc \equiv_\iota bac \equiv_\iota acb$, $ab \equiv_\iota ba$

hence

$d([abc], [ab]) = 1/4$
$d([ab], [cd]) = 1$
$d([abc], [cd]) = 1/2$

It is easy to see that the triangular inequation does not hold. It has recently come to our knowledge that in [LuG87] a metric was defined for event structures. Future work will show if such an approach could also be applied to the domain of traces. It should, however, be emphasized that, since the Scott topology is non-Hausdorff, a quasi-metric would have to be used in order to achieve convergence that corresponds to the Scott closure.

Positive Approach

The approach presented here has been negative, rather than positive; in other words, the work has concentrated on ways of *excluding* those behaviours that are not fair, rather than *generating* those behaviours only that are fair in some sense.

It is an interesting question whether, given a fairness property, it is possible to determine a set of rules that generate those behaviours only which satisfy the given fairness property. When such rules are applied to verify properties of systems, fairness constraints would automatically be incorporated. We have discovered the following limitation: although asynchronous transition systems accept infinitary trace languages, they only accept languages that are closed and prefix-closed. It has become clear to us that fairness properties correspond to languages that are ideals, but are not, in general, closed in the sense of containing adherence. The notion of maximality has proved helpful in excluding undesirable behaviours. Unfortunately, restricting infinite behaviours to maximal ones is not sufficient when imposing synchronisation fairness as, in this case, it is necessary also to exclude those which are maximal.

Temporal Logic

The topological characterization of behavioural properties has enabled us to formally state and characterize, but not necessarily establish, certain properties of asynchronous systems. Temporal logic can be used to bridge this gap, and, as such, provide a formalism for reasoning about fairness and liveness.

Process Algebra and Bisimulations

An issue we have not elaborated on in the thesis is what kind of process algebra the model gives rise to. We have strong reasons to believe that such an algebra could be defined. In fact, a basic framework for such a finitary process algebra has been formulated in [Maz84b], where the operations of sequential composition and synchronisation were defined. These issues have not been considered here as our main concern were infinitary aspects of trace theory.

A further difficulty is that the definition of fairness properties does not allow for dynamic processes, that is recursion over concurrency as found in most process algebras, for example CCS. This is a result of an earlier restriction of the set of action labels A to a finite set. A possible solution to this problem would involve enhancing the model to allow for a countable set of action labels, which, in consequence, would enable a countable number of processes.

A notion which does not seem to have a counter-part in trace theory is that of bisimulation. Bisimulation was introduced in [Par81] as an equivalence notion which could be used in situations in which equivalence of systems in terms of languages they recognise is not appropriate. It is not clear how weak bisimulations could be defined for asynchronous transition systems (strong bisimulations were formulated in [Shi88c]). The difficulty seems to lie in the way the local view could be put in the context of a global view.

CCS and Asynchronous Transition Systems

It has already been mentioned that CCS determines an asynchronous transition system. As a result, it is possible to provide CCS with asynchronous semantics. We shall briefly

sketch how this could be achieved. The main problem with CCS is that it is ambiguous; for example $p = ap' + ap''$, where $p' \neq p''$, is an example of an ambiguous expression. Thus, some systematic technique has to be used in order to disambiguate transitions. It has been shown in [BoC88] that a labelling with proofs could be used for this purpose (a similar labelling was developed in [CoS87]). A proof of a transition $p \rightarrow^a p'$ is an indication of how the action a is obtained from the term p. Roughly speaking, a proof is a path which leads to an *outermost* subterm (for communication the proof is a path leading to a pair of complementary subterms). The above labelling determines the independency relation between proof terms. We assume θ denotes such proof, γ_a denotes the guard a, $\pi_1(\theta)$ and $\pi_2(\theta)$ denote the left and right operand of a parallel composition respectively, $\delta(\theta,\theta')$ denotes communication, $\sigma_1(\theta)$ and $\sigma_2(\theta)$ - left and right component of a sum, and finally $\rho_a(\theta)$ denotes restriction with a.

The independency relation ι is the least symmetric one which satisfies the following.

(A1) $i \neq j \Rightarrow \pi_i(\theta)\ \iota\ \pi_j(\theta')$

(A2) $\theta\ \iota\ \theta' \Rightarrow \pi_1(\theta)\ \iota\ \delta(\theta',\theta'')$
$\theta\ \iota\ \theta' \Rightarrow \pi_2(\theta)\ \iota\ \delta(\theta'',\theta')$

(A3) $\theta\ \iota\ \theta' \Rightarrow \pi_i(\theta)\ \iota\ \pi_i(\theta')$
$\theta\ \iota\ \theta' \Rightarrow \sigma_i(\theta)\ \iota\ \sigma_i(\theta')$
$\theta\ \iota\ \theta' \Rightarrow \rho_a(\theta)\ \iota\ \rho_a(\theta')$

(A4) $\theta_1\ \iota\ \theta_1' \wedge \theta_2\ \iota\ \theta_2' \Rightarrow \delta(\theta_1,\theta_2)\ \iota\ \delta(\theta_1',\theta_2')$.

It is easy to see that ι is irreflexive. We then proceed to define an asynchronous transition system by taking the set of all closed CCS terms as the set Q of states, the set of proofs as the set A of action labels, and the relation ι as the the independency. Finally, CCS transition rules need to be suitably modified to determine the transition relation $\rightarrow$ of the asynchronous transition system in question.

Quantitative Methods for Fairness

Only qualitative methods for dealing with fairness have been directly discussed. We have not, for example, outlined ways by which the relative frequency of choice of particular transitions could be controlled (probabilistic fairness). Also, unbounded delay has been

assumed. Probabilistic fairness and bounded delay fairness seem to be expressible in terms of the strength predicates and would, therefore, be a worthwhile subject for investigation. The immediate applicability of such results to real-time and fault-tolerant systems should be apparent.

Final Remarks

We would like to stress that, in contradiction to the statement made in [Dij88], our research has shown that fairness properties can be formally characterized and stated. Although undetectable by finite experiments, fairness nevertheless provides a useful abstraction.

Appendix

CCS Summary

Let $A = \Delta \cup \overline{\Delta}$ such that $\Delta \cap \overline{\Delta} = \varnothing$ be a set of *actions*. We require Δ, $\overline{\Delta}$ are in bijection, that is $\overline{a} \in \overline{\Delta}$ stands for the *co-action* of $a \in \Delta$. Let $M = A \cup \{\tau_a \mid a \in \Delta\}$ be the set of *moves*.

We assume a,b,c range over A, m,n range over M, and X,Y are process variables. The syntax of CCS expressions extended with process composition is:

$p ::= X \mid NIL \mid mp \mid p+p \mid fix\ X.p \mid p\backslash a \mid p \| p \mid p;p.$

Semantics is as follows:

(i) $mp \rightarrow^m p$

(ii) $p \rightarrow^m p'$ implies $(p+q) \rightarrow^m p'$, $(q+p) \rightarrow^m p'$

(iii) $p[fix\ X.p\backslash X] \rightarrow^m p'$ implies $fix\ X.p \rightarrow^m p'$ where $[.\backslash .]$ denotes substitution.

(iv) $p \rightarrow^m p'$, $m \notin \{a, \overline{a}\}$ implies $p\backslash a \rightarrow^m p'\backslash a$

(v) $p \rightarrow^m p'$ implies $(p \| q) \rightarrow^m (p' \| q)$, $(q \| p) \rightarrow^m (q \| p')$

(vi) $p \rightarrow^a p'$, $q \rightarrow^{\overline{a}} q'$ implies $(p \| q) \rightarrow^{\tau} a\ (p' \| q')$

(vii) $p \rightarrow^m p'$, $p' \rightarrow^n p''$ implies $p;q \rightarrow^m p';q$
$p \rightarrow^m p'$, $\forall\ n \in M$: $p' \not\rightarrow^n p''$ implies $p;q \rightarrow^m q$.

Bibliography

[AaR88] Aalbersberg I.J., Rozenberg G., Theory of Traces, *Theoretical Computer Science* **60** (1988) 1-82.

[AFG88] Attie P., Francez N., Grumberg O., Fairness and Hyperfairness in Multi-party Interactions", submitted to *Distributed Computing*.

[AFK87] Apt K.R, Francez N., Katz S., Appraising Fairness in Languages for Distributed Programming, in: *Proceedings of the 14th ACM Symposium on Principles of Programming Languages* (1987). To appear in *Distributed Computing*.

[AlS85] Alpern B., Schneider F.B., Defining Liveness, *Information Processing Letters* **21** (1985) 181-185.

[ApF84] Apt K.R., Francez N., Modelling the distributed termination convention in CSP, *ACM Transactions on Programming Language and Systems* **6**, 3 (1984) 370-379.

[ApO84] Apt K.R., Olderog E-R., Transformations Realizing Fairness Assumptions for Parallel Programs, in: *Proceedings of Symposium on Theoretical Aspects of Computer Science* (1984), Lecture Notes in Computer Science **166** (Springer, 1984).

[Ash75] Ashcroft E.A., Proving Assertions about Parallel Programs, *Journal of Computer And System Sciences* **10** (1975) 110-135 .

[Bed87] Bednarczyk M., *Models for Parallelism*, PhD thesis, University of Sussex (1987).

[BeK86] Bergstra J.A., Klop J.W., Algebra of communicating processes, in: *Proceedings of the CWI Symp. Math. & Comp. Sci.*, de Bakker J.W., Hazenwinkel M., Lenstra J.K., eds., Amsterdam (1986).

[Bes84a] Best E., Fairness and Conspiracies, *Information Processing Letters* **18** (1984) 215-220.

[Bes84b] Best E., erratum, *Information Processing Letters* **19** (1984) 162.

[BoC88] Boudol G., Castellani I., Permutation of transitions: an event structure semantics for CCS and SCCS, in: *REX School/Workshop on Linear Time, Branching time and Partial Order in Logics and Models for Concurrency* (Nordwijkerhout, 1988). To be published in Lecture Notes in Computer Science.

[BoN79] Boasson L., Nivat M., Adherences of Languages, *Journal of Computer and System Sciences* **20** (1980) 285-309.

[BHR84] Brookes S.D., Hoare C.A.R., Roscoe W., A Theory of Communicating Sequential Processes, *Journal of the ACM* **31**, 3 (1984) 560-599.

[BKP84] Barringer H., Kuiper R., Pnueli A., Now You May Compose Temporal Logic Specifications, in: *Proceedings, 16th Symposium on Theory of Computing* (1984) 51-63.

[BMO87] de Bakker J.W., Meyer J.-J. C., Olderog E.-R., Infinite Streams and Finite Observations in the Semantics of Uniform Concurrency, *Theoretical Computer Science* **49** (1987) 87-112.

[Car87] Carstensen H., in: *Proceedings, STACS 87*, Brandenburg F.J., Vidal-Naquet G., Wirsing M., Lecture Notes in Computer Science **247** (Springer, 1987).

[CaH87] Cavalli A.R., Horn F., Proof of specification properties by using finite state machines and temporal logic, in: *Proceedings of the IFIP WG 6.1 7th International Conference* (1987).

[CaV84] Carstensen H., Valk R., Infinite Behaviour and Fairness in Petri Nets, in: *Advances in Petri Nets 84*, G.Rozenberg, ed., Lecture Notes in Computer Science **188** (Springer, 1984).

[ChM88] Chandy K.M., Misra J., Another View of 'Fairness', *Software Engineering Notes* **13**, 3 (July 1988) 20.

[ClG87] Clarke E.M., Grumberg O., Research on Automatic Verification of Finite-State Concurrent Systems, *Ann. Rev. Comput. Sci.* **2** (1987) 269-290.

[CoS84] Costa G., Stirling C., A Fair Calculus of Communicating Systems, *Acta Informatica* **21** (1984) 417-441.

[CoS87] Costa G., Stirling C., Weak and Strong Fairness in CCS, *Information and Computation* **73** (1987) 207-244.

[Dij76] Dijkstra E.W., *A Discipline of programming*, Prentice-Hall (1976).

[Dij88] Dijkstra E.W., Position Paper on Fairness, *Software Engineering Notes* **13**, 3 (April 1988) 18-20.

[EmH86] Emerson E.A., Halpern J.Y., 'Sometimes' and 'Not Never' Revisited: On Branching versus Linear Time Temporal Logic, *Journal of the ACM* **33**, 1 (1986) 151-178.

[Fau87] Fauconnier, Semantique Asynchrone et Comportements Infini en CSP, *Theoretical Computer Science* **4** (1987) 277-298.

[Fra86] Francez N., *Fairness*, Springer-Verlag (New York, 1986).

[GPS80] Gabbay D., Pnueli A., Shelah S., Stavi J., On the Temporal Analysis of Fairness, in: *Proceedings of the 7th ACM Symposium on Principles of Programming Languages* (1980).

[GFK86] Grumberg O., Francez N., Katz S., A Complete Rule for Equifair Termination, *Journal of Computer and System Sciences* **33** (1986) 313-332.

[GHK80] Gierz G., Hofmann K.H., Keimel K., Lawson J.D., Mislove M., Scott D., *A Compendium of Continuous Lattices*, Springer (1980).

[Hal60] Halmos J.R., *Naïve Set Theory*, Van Nostrand (1960).

[Hen87] Hennessy M., An Algebraic Theory of Fair Asynchronous Communicating Processes, *Theoretical Computer Science* **49** (1987) 121-143.

[Hen83] Hennessy M., Modelling Finite Delay Operators, Technical Report CSR-153-83, University of Edinburgh (1983).

[Hoa78] Hoare C.A.R., Communicating Sequential Processes, *Communications of the ACM* **21**, 9 (1978) 666-677.

[Hoa84] Hoare C.A.R., *Communicating Sequential Processes*, Prentice-Hall (1984).

[KaM69] Karp R.M., Miller R.E., Parallel Program Schemata, *Journal of Computer and System Sciences* **3** (1969) 147-195.

[KaL76] Karp R.A., Luckham D.C., Verification of fairness in an implementation of monitors, in: *2nd International Conf. on Software Engineering* (IEEE, 1976)

[KdR83] Kuiper R., de Roever W.P., Fairness assumptions for CSP in a temporal logic framework, in: D. Bjorner, ed., *Proceedings of TC.2 Working Conference on the Description of Programming Concepts* (North-Holland 1983).

[Kel76] Keller R.M., Formal Verification of Parallel Programs, *Communications of the ACM* **19** no.7, pp 371-384 (1976).

[Kur66] Kuratowski K., *Topology*, Academic Press (1966).

[Kwi88a] Kwiatkowska M.Z., Modelling Concurrency Using Ambiguous Asynchronous Transition Systems, Technical Report No. 9, University of Leicester, Department of Computing Studies (1988).

[Kwi88b] Kwiatkowska M.Z., Event Fairness in Asynchronous Transition Systems, Technical Report No. 10, University of Leicester, Department of Computing Studies (1988).

[Kwi88c] Kwiatkowska M.Z., A Survey of Fairness Notions, Technical Report No. 14, University of Leicester, Department of Computing Studies (1988).

[Kwi89] Kwiatkowska M.Z., A Survey of Fairness Notions, to appear in *Information and Software Technology*.

[Kwo79] Kwong Y.S., On the Absence of Livelocks in Parallel Programs, in: *Semantics of Concurrent Computation, Proceedings* (1979), G.Kahn, ed., Lecture Notes in Computer Science **70** (Springer, 1979).

[Lam77] Lamport L., Proving the Correctness of Multiprocess Programs, *IEEE Transactions on Software Eng.* **SE-3**, 2 (1977) 125-143.

[Lam86] Lamport L., The mutual exclusion problem. II. Statement and solutions, *Journal of the ACM* **33** (86) 327-348.

[LPS81] Lehman D., Pnueli A., Stavi J., Impartiality, Justice and Fairness: The Ethics of Concurrent Termination, in: *Proceedings, Automata, Languages and Programming* (1981), S.Even, O.Kariv, eds., Lecture Notes in Computer Science **115** (Springer, 1981).

[LSB79] Lauer P.E., Shields M.W., Best E., Design and Analysis of Highly Parallel and Distributed Systems, in: *Abstract Software Specifications, Proceedings*, D. Bjorner, ed., Lecture Notes in Computer Science **86** (Springer, 1980).

[LuG87] Lugen, R., Goltz U., Non-Interleaving Semantic Model for Non-Deterministic Concurrent Processes, Technical Report 87/15, RWTH Aachen, Fachgruppe Informatik (1987).

[MaP81] Manna Z., Pnueli A., Temporal verification of concurrent programs, in: Boyer R.S., J. Strother Moore, eds., *The Correctness Problem in Computer Science* (Academic Press, 1981) 215-173.

[Mar81] Martin A.J., An axiomatic definition of synchronisation primitives, *Acta Informatica* **16**, 2 (1981) 219-235.

[Maz77] Mazurkiewicz A., Concurrent Program Schemes and Their Interpretations, DAIMI Report PB - 78, Aarhus University (1977).

[Maz84a] Mazurkiewicz A., Traces, Histories, Graphs: Instances of a Process Monoid, in: *Proceedings, Mathematical Foundations of Computer Science* (1984), Chytil M.P., Koubek V., eds., Lecture Notes in Computer Science **176** (Springer, 1984)..

[Maz84b] Mazurkiewicz A., Semantics of Concurrent Systems: A Modular Fixed-Point Trace Approach, in: *5th European Workshop on Applications and Theory of Petri Nets* (1984).

[Maz88] Mazurkiewicz A., Basic Notions of Trace Theory, in: *REX School/Workshop on Linear Time, Branching time and Partial Order in Logics and Models for Concurrency* (Nordwijkerhout, 1988). To be published in Lecture Notes in Computer Science.

[Mer86] Merceron A., Fair Processes, in: *Proceedings, 7th European Workshop on Applications and Theory of Petri Nets* (Oxford, 1986).

[Mil80] Milner R., *A calculus for communicating systems*, Lecture Notes in Computer Science **92** (Springer, 1980).

[Mil88] Milner R., Operational and Algebraic Semantics of Concurrent Processes, Technical Report No. ECS-LFCS-88-46, University of Edinburgh, Department of Computer Science (1988).

[MOP88] Mazurkiewicz A., Ochmanski E., Penczek W., Concurrent Systems and Inevitability, to appear in *Theoretical Computer Science*.

[OwG76] Owicki S., Gries D., An Axiomatic Proof Technique for Parallel Programs, *Acta Informatica* **6** (1976) 319-340.

[OwL82] Owicki S., Lamport L., Proving Liveness Properties of Concurrent Programs, *ACM Transactions on Programming Languages and Systems* **4**, 3 (1982) 455-495.

[Par81] Park D., Concurrency and Automata on Infinite Sequences, in: *Proceedings of the 5th GI Conference on Theoretical Computer Science*, P. Deussen, ed., Lecture Notes in Computer Science **104** (Springer, 1981).

[Par80] Park D., On the Semantics of Fair Parallelism, in: *Abstract Software Specifications, Proceedings* (1979), D. Bjorner, ed., Lecture Notes in Computer Science **86** (Springer, 1980).

[Par85] Parrow J., Fairness Properties in Process Algebra with Applications in Communication Protocol Verification, PhD Thesis, Department of Computer Systems, Uppsala University (1985).

[Plo82] Plotkin G., A Powerdomain for Countable Non-determinism, in: *Automata, Languages and Programming, 9th Coll*, Lecture Notes in Computer Science **140** (Springer, 1982).

[Pnu83] Pnueli A., On the extremely fair treatment of probabilistic algorithms, in: *Proceedings of the 15th ACM Symposium on Theory of Computing* (1983) 278-290.

[Pnu86] Pnueli A., Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends, in: *Current Trends in Concurrency*, de Bakker, de Roever, Rozenberg, eds., Lecture Notes in Computer Science **224** (Springer, 1986).

[PnZ86] Pnueli A., Zuck L., Verification of multiprocess probabilistic protocols, *Distributed Computing* **1** (1986) 53-72.

[PRW87] Priese L., Rehrmann R., Willecke-Klemme U., An Introduction to the Regular Theory of Fairness, *Theoretical Computer Science* **54** (1987) 139-163.

[QuS83] Queille J.P., Sifakis J., Fairness and Related Properties in Transition Systems - A Temporal Logic to Deal with Fairness, *Acta Informatica* **19** (1983) 195-220.

[Rei85] Reisig W., *Petri Nets, An Introduction*, Springer (1985).

[Ros88] Roscoe A.W., Analysing infinitely branching transition systems, preprint (1988).

[RoT86] Rozenberg G., Thiagarajan P.S., Petri nets: basic notions, structure, behaviour, in: *Current Trends in Concurrency*, de Bakker, de Roever, Rozenberg, eds., Lecture Notes in Computer Science **224** (Springer, 1986).

[ScG87] Scott D., Gunter C.A., Semantic Domains, to appear in Handbook of Theoretical Computer Science, North-Holland.

[Shi85a] Shields M.W., Deterministic Asynchronous Automata, in: *Formal Methods in Programming* (North-Holland, 1985).

[Shi85b] Shields M.W., Concurrent Machines, *The Computer Journal* **28**, 5 (1985) 449-466.

[Shi88a] Shields M.W., Behavioural Presentation, in: *REX School/Workshop on Linear Time, Branching time and Partial Order in Logics and Models for Concurrency* (Nordwijkerhout, 1988). To be published in Lecture Notes in Computer Science.

[Shi88b] Shields M.W., private communication.

[Shi88c] Shields M.W., *Elements of a Theory of Parallelism*, to be published.

[Sto77] Stoy J. E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, The MIT Press (1977).

[Smy83] Smyth M.B., Power Domains and Predicate Transformers, a topological view, in: *Automata, Languages and Programming, Proceedings*, Lecture Notes in Computer Science **154** (Springer, 1983).

[vGl86] van Glaabbek R.J., Bounded non-determinism and induction principle in process algebra, Technical Report CS-R8634, Centrum foor Wiskunde en Informatica, Amsterdam (1986).

[Win86] Winskel G., Event Structures, Technical Report No.95, Computer Laboratory, Cambridge University (1986).