Survey of fairness notions

M Z Kwiatkowska

Fairness has been examined from different viewpoints and in varied semantic models, for example, CCS, guarded commands, Petri nets, and automata. Classification of fairness notions has been proposed in many formalisms, which have 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 interrelationships 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.

The paper 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 standard methods when applied to deal with fairness.

fairness, fairness behaviour, fairness constraints, concurrency, nondeterminism

A system is said to be correct if it satisfies a given specification, where the specification is a list of properties. To show that the system has a given property, typically an abstract model is used 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. It has long been recognised, however, that this is not the case with concurrent (or nondeterministic) 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.

Consider an example of an airline reservation system. The system consists of a waiting list of booking requests and a passenger list, each list being manipulated by a manager process capable of dealing with one customer at a time. Customers who make booking requests are first registered on the waiting list, from which they enter the booking process, in some order, one by one. This 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 at both hardware and software levels, that is, all three processes mentioned 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. It might be thought that this would be guaranteed by the truly concurrent implementation as the processes do not have to wait for their time-slice to proceed. However, the reality is different. Imagine that 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 yet another flight. For some reason, B's request, rather than A's, is chosen when C has exited the booking process. C decides to book yet another flight too, however, 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, the system could have been proved partially correct using an inductive argument based on invariants.) The only concern is that customer A has not made satisfactory progress, although he was continuously ready to do so. It cannot be shown that A's request is eventually serviced unless explicit assumptions are made 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

Department of Computing Studies, University of Leicester, University Road, Leicester LE1 7RH, UK

 0950-5849/89/070371-16 $3.00

such properties under fairness constraints, transfinite induction (see section 'Proving properties under fairness constraints' for explanation) may be needed. It might be argued 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 multiprogramming 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, the concern should be with ways that help to 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, nondeterministic) systems. The work in this area concentrates on building proof systems for proving properties under fairness constraints (like the 'guaranteed service' mentioned above) and the development of proof techniques[1–6]. Other major issues include examining the adequacy of existing formalisms when expressing fairness and investigating possible extensions[7–9]. In addition, there is some research concerned with the classification (e.g., as a hierarchy) of fairness properties[9–13].

To the author's knowledge, there have been few attempts[9] to provide a formal definition of 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, fairness notions shall be discussed through examples, rather than through a formal comparative study. An informal survey like this should precede any formal discussion and should provide the necessary background, encouragement, and direction. A preliminary version of this paper appeared in Kwiatkowska[14].

DEFINING FAIRNESS

Before presenting a taxonomy of specific fairness properties, the main concerns that guided previous research when defining fairness, are summarized.

Fairness properties in concurrent and nondeterministic systems shall be informally discussed. By a system shall be meant 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 the guarded commands of CSP[15]. A sequence of states with associated actions is called, for the purpose of this paper, 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, or a partially ordered set) to represent the behaviour of a system, and hence to be considered a viable semantics. The presentation here will ignore those differences, as detailed discussion is beyond the scope of this paper (see Kwiatkowska[9] for more details).

Note that when considering fairness, it is necessary to include both infinite and finite computations.

Intuitive definition

The motivation behind any notion of fairness is 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, however, it may be necessary to distinguish between fair and unfair computations. Intuitively, fairness is a property of computations that can be expressed as follows: no component of the system that becomes possible sufficiently often should be delayed indefinitely.

This is a general statement that brings together many fairness properties known from literature. To obtain a specific fairness property it is necessary to say explicitly what is meant by a 'system component', 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 some kinds of components, and what it means for each to become possible, are given in Table 1.

As most models have an underlying state-transition system structure, the notion of a component becoming possible shall often be identified with it being enabled. The notion of a component becoming possible may vary depending on the semantics chosen. 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 [9, 16–18].

Table 1. Notion of system component becoming possible: some examples

Component	Becomes possible if
(Concurrent) process	Some action of the process is enabled
Event	Event can occur
Synchronization event	Processes can synchronize
Channel communication	Processes can communicate on channel
Guard	Some guard in a nondeterministic program evaluates to true
Transition	Transition becomes enabled in a given state
State	State is immediately reachable from a given state

Table 2. Most common variations on strength of fairness

Sufficiently often	Corresponding strength
Constraint free	Unconditional fairness
Infinitely often	Strong fairness
Almost always	Weak fairness

Table 2 gives the most common variations on the strength of fairness. The terms in Table 2, although informal, are widely used and will be explained later. 'Almost always'[6,19] means at every step after some point in time and can also be described as 'continuously from some point onwards'[5,8].

The intuitive definition of fairness can be rephrased in the following, perhaps more familiar, form: if a system component becomes possible sufficiently often then it proceeds infinitely often. Note that the intuitive 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[20]. Another interesting class of fairness properties are probabilistic fairness properties[4].

Concurrency fairness versus fairness of choice

It is possible to view fairness either as an issue fundamentally to do with concurrency or nondeterminism. This leads to a distinction between concurrency fairness and fairness of choice. 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 nondeterministic. Both P and Q never terminate and print infinite sequences that consist of a's and b's. If fairness is defined as a property that states that no concurrent process should be delayed indefinitely (concurrency fairness), then a computation that allows an infinite sequence of a's (denoted a^{ω}) is unfair with respect to this notion because it ignores process B (and likewise any execution that allows only a finite number of b's). When considering program Q, however, a^{ω} must be considered concurrency fair (and, indeed, any computation that allows only a finite number of b's); program Q does not contain any concurrent processes, and hence no concurrent process in Q has been indefinitely delayed!

Nothing prevents definition of fairness with respect to the choice of nondeterministic guards (fairness of choice). Thus an execution is fair with respect to choice if no nondeterministic guard is ignored forever. Notice that, for program Q, this means disallowing computations that lead to either a or b being printed only a finite number of times. In program P, on the other hand, all computations are fair with respect to choice because there is no nondeterministic choice in P, hence no guard has been discriminated against!

Concurrency fairness and fairness of choice (also called 'fairness in selection'[8] or 'conflict resolution fairness'[21]) are independent notions, although they are sometimes identified. This confusion is caused by the fact that concurrency is often reduced to nondeterministic interleaving, in which case concurrency fairness corresponds to fairness of choice for a particular scheduler. Note that to distinguish these two notions of fairness it is necessary to consider a formalism that allows for a distinction between concurrency and nondeterminism.

Fairness of choice[3,5] and concurrency fairness[2,6,7] are discussed elsewhere.

Fairness and granularity level

Fairness properties are severely affected by the choice of granularity level, that is, the kind of system component under consideration, e.g., process, event, transition, and state. For a given system, they typically result in fairness notions 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 as 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. If it is assumed that 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, 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 that contain 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 nondeterministic 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 redefine the notion of an event so that event fairness coincides with process fairness (after all, it has been noted[19] that event fairness depends on what constitutes an event). A more orthodox view might be taken, however, 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 related) notions.

Event fairness has been defined[19] for a concurrent 'while' language. Process fairness for a (shared-memory) concurrent programming language[2,8] and for CCS[6,22] has been discussed. The relationship of process and event fairness in a noninterleaving semantic model has been considered[9].

In the context of distributed models that allow interprocess communication, e.g., over channels, granularity of fairness can be further refined by defining channel fairness and process communication fairness[10] (not necessarily on the same channel). It can be shown that, together with process fairness, these notions form a strict hierarchy. (That is, the computations that are fair with respect to one notion are strictly contained in the set of computations that are fair with respect to another one.)

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 it is possible to distinguish between transition fairness[3], in the sense that no transition that is infinitely often enabled should be ignored indefinitely, and fair reachability from states[3], in the sense that no state that is immediately reachable infinitely often should be delayed indefinitely.

Consider the following example (adapted from Francez[5]). The states are identified with the values of x, the transitions are the assignment statements, and the guards determine if the transitions are enabled in a given state:

```
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 second guard) that takes both transitions infinitely often, thus no transition has been ignored indefinitely (transition fairness). 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, both guards are no longer enabled and the program terminates. Thus P terminates under fair reachability from states but does not under transition fairness constraints.

Fair reachability from states and transition fairness are independent notions. Here their strong versions only have been introduced, although other strengths may also be considered[5].

Fairness and liveness

It has long been recognised[23] that fairness affects liveness properties of programs. Liveness has been defined as a class of properties that state that something good will happen during program execution[23]. This is in contrast with safety[23], which is pronounced as nothing bad will happen. Examples of liveness are program termination (the 'good thing' is that 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 nondeterministic or concurrent programs; certain programs cannot be proved correct with respect to a given liveness property unless explicit assumptions are made 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 synchronize. Process P terminates only if A and B eventually synchronize. Note that P does not terminate if no arbitration is present; if, however, a 'reasonable' scheduling algorithm is assumed, P will terminate. Termination of P cannot be formally proved unless the assumption is incorporated in the proof 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: unless a process has terminated, it will proceed infinitely often[24]. The author believes that this formulation is a fairness notion called process liveness, whereas the stated meaning refers to a more general class of properties.

A topological characterization of liveness has been presented[25]. A number of proof techniques for liveness have been introduced, namely, the proof lattice method[26] and the well founded sets method[1,8]. The relationship of fairness and liveness has also been considered[9].

Negative versus positive approach

The question of how to discriminate between fair and unfair computations has so far been avoided. Note that this cannot be achieved (in a finite number of steps) by examining every finite prefix of some computation. Also, many existing semantic models do not make any pro-

visions for fairness; it is generally accepted that without fairness 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[2]. The lower level admits all possible computations, whether fair or unfair. Then, at the higher level, some methods are invoked for excluding unfair computations. An example of such a method is to introduce proof rules, one for each fairness notion, which are then 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 constraints[8,27]. The motivation for the negative approach is to produce correctness proofs with respect to a given scheduling policy, which could then be enforced at the implementation stage.

An alternative approach is to generate only fair computations of a given system, rather than exclude the unfair ones from all admissible computations. This is the essence of the positive approach, which deals with one semantic level. A set of rules generating weakly and strongly process fair computations for CCS has been developed[6,22] by extending the CCS calculus to a labelled calculus and modifying the proof rules. There are other positive approaches, based on random assignment[19,28]. The positive approach could be viewed as producing a set of guidelines for an acceptable scheduler from the point of view of the semantics of the system.

TAXONOMY OF FAIRNESS NOTIONS

Some of the most widely known fairness notions are now presented in more detail.

Unconditional fairness

Unconditional fairness (also called impartiality) has originally been defined for processes[2]. It can be summarized as follows: every process proceeds infinitely often. In other words, every unconditionally fair computation must admit an infinite number of occurrences of the actions of each process. An example of unconditional fairness has already been considered in the section 'Concurrency fairness versus fairness of choice' (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:

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

Process B, unlike process A, terminates after executing its action. It is unreasonable, therefore, to expect process B to proceed infinitely often. Intuitively, the set of admissible computations should include only those computations that generate an infinite number of a's interleaved with one b (i.e., a^*ba^ω). However, this computation is not unconditionally fair because b does not occur infinitely often!

Thus unconditional fairness should only be used in the context of nonterminating processes. (Excluding distributed termination does not reduce the expressive power of some models, e.g., CSP[29], so unconditional fairness will often be sufficient.) Likewise, processes that have not yet terminated but have become disabled, for example, by waiting to synchronize with another process, will not be correctly handled by unconditional fairness.

Unconditional process fairness properties in a noninterleaving model for concurrency have been shown to form a lattice[9]. Unconditional fairness may also be defined with respect to nondeterministic choice[5].

Process liveness[24] is a modification of unconditional fairness to allow for the fact that processes may terminate. (Process liveness should not be confused with liveness as 'something good will happen'[23], which is a class of more general properties.) It can be expressed as follows: unless a process has terminated, it will proceed infinitely often.

Now, in the program P above, process liveness will admit only the computations a^*ba^ω. However, process liveness is still a very crude property. Note that the above definition can be rephrased 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 synchronization; it is possible for a process to become (temporarily or permanently) disabled because it is waiting to synchronize with another process that refuses to do so.

Unconditional fairness is known in formal languages as 'fairmerge', which was introduced for ω-regular languages (i.e., extensions of regular languages, by means of the ω-iteration operator, onto languages of finite or infinite sequences over a given alphabet)[28]. Fairmerge of two such languages A and B is a language A||B that contains only those sequences that 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 noncommunicating, nonsynchronizing concurrency, that is, systems that consist of concurrent processes whose actions are totally independent. Some aspects of fairmerge have been considered for process algebras[30]. Fairmerge is not adequate to express synchronization fairness, but it can be suitably extended[31].

Weak fairness

Weak process fairness (also called 'justice')[2,8] is a property that takes into account the fact that processes may become disabled. A process becomes possible (or enabled) if at least one 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 onwards then it eventually proceeds.

This definition excludes all 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.

As 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.

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 synchronize, whereas process B can repeatedly choose between synchronization with A and the internal action. P terminates only if A and B synchronize (the second guard then becomes false). If weak process fairness is assumed 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 cost of a possible implementation[8]; it is automatically guaranteed on a truly concurrent system, whereas for multiprogramming 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 nondeterministic guards, which may be phrased as 'any guard that evaluates to **true** from some point on will eventually be chosen'. 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 has been considered [6]. Weak process fairness properties have been defined for noninterleaving concurrency[9]. Weak process, channel communication, and process communication fairness for CSP have been discussed[10] and an overview of weak fairness of choice presented[5]. Weak fairness can also be defined at the level of events[19], transitions[3], and states[3].

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. Therefore the assumption of a process becoming enabled continuously from some point on is relaxed to becoming enabled infinitely often. Thus a notion of strong process fairness (also called fairness[2]) is arrived at that 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 becomes disabled, strong process fairness coincides with weak process fairness.

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 synchronize with B, but B can accept the synchronization 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 synchronize. An infinite computation that allows 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. 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 nondeterministic guards, which may be described as any guard that evaluates to **true** infinitely often will be chosen

infinitely often. As an example, 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 has been considered[6]. Strong process fairness properties have been defined for noninterleaving concurrency[9]. Strong process, channel communication, and process communication fairness in CSP have been discussed[10] and an overview of strong fairness of choice given[5]. Strong fairness may also be defined at the level of events[19], transitions[3], and states[3].

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 it is possible to arrive at a notion of equifairness[5,20], the motivation for which is to give each 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. Consider the following example (adapted from Francez[5]):

```
Q = (x, y := 0; z := 1;
     do z > 0                  → x := x + 1
     □ z > 0                   → y := y + 1
     □ z > 0 and x = y → z := z − 1
     od)
```

This program does not terminate under the assumption of strong fairness of choice of nondeterministic 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.

Strong equifairness is strictly stronger than strong fairness of choice of nondeterministic guards. (In other words, the set of equifair computations is strictly contained in the set of computations that are fair with respect to strong fairness of choice.) The same holds for weak and unconditional equifairness[5]. Equifairness is usually applied to nondeterministic guards, although it seems feasible to apply it in other situations as well, for example, process synchronization.

Probabilistic fairness

Fairness has also been defined as a probabilistic property[4,32]. A computational model, based on a state-transition system with probabilities, is assumed attached to transitions. A determinate transition (i.e., nonprobabilistic) 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, each with probability α_i attached; every τ^i is called a mode of the transition τ. It is required that $\alpha_1 + \alpha_i + \cdots + \alpha_k = 1$. 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[32] that 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[4] to verify multiprocess protocols.

Other notions

Many more extensions or generalizations of fairness have been introduced. Generalized fairness[5] 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[3] as a replacement for state and transition fairness is a similar notion; a computation is considered unfair if there is a predicate over states that is reachable infinitely often but states satisfying this predicate are taken only finitely often. Relativized fairness[3] 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 [33,34]. Hyper-fairness[35] is a conspiracy-resistant notion adequate for multiparty operations (the reason for this distinction is that some fairness notions allow only pairwise communication; multiparty communication can be found in some models, e.g., TCSP[36] and programming languages like ADA).

Related notions

Several notions have been introduced that can be in some way related to fairness; in fact, in many cases, the author believes they are fairness properties.

Finite delay

Finite delay property was originally formulated by Karp and Miller[16], 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 may remain enabled, or infinite, in which case it is required to satisfy finite delay property. The finite delay property can be described informally as: if some transition is permanently enabled from some point on then it is eventually taken.

The author believes the intention of this definition was to impose finite delay for independent, that is, concurrent, actions, rather than nondeterministic choices. In fact, a subclass of systems (determinate systems) that are concurrent but disallow nondeterminism has been characterized[16]; for such systems, finite delay corresponds to concurrency fairness. When adapting this property to the more general class of nondeterminate systems, it seems necessary to distinguish whether a transition remains enabled in the context of concurrency or nondeterminism; otherwise, the resulting property may be considered too strong[9], as it would impose concurrency fairness together with fairness of choice.

The finite delay property is a fairness property that reflects the minimum constraint on sequences of transitions/states to be admissible as sequentializations of a concurrent behaviour, and it is too weak to model synchronization fairness. Finite delay only excludes certain infinite computations; hence, as such, it affects properties like termination and equivalence but bears no relation to partial correctness. It is present, as maximality of computations, in many noninterleaving models for concurrency based on causality[9,17,18,21,37,38]. The interleaving models, like CCS[39] or TCSP[36], have often been criticised for not having a finite delay property. A finite delay operator has been proposed[40], but it is not clear how this approach relates to that of Karp and Miller[16].

Livelocks

The term 'livelock' was coined by Ashcroft[41], who extended the Floyd assertion method onto parallel programs. Ashcroft observed that, although, using induction, he could 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 did 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 showed 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.

The motivation for introducing the notion of a livelock was to impose fairness constraints. Livelocks can also be viewed as a liveness property[23] (the 'good thing' would be the customer making progress). Livelocks have been examined in a Petri-net setting[42].

Unbounded nondeterminism

Fairness is often discussed in the context of unbounded nondeterminism, which was first raised by Dijkstra[43]. This phenomenon arises in the context of guarded commands:

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

Under the assumption of (weak) fairness of choice, P would always terminate and yet produce any, that is, unbounded, natural number. This seems to contradict the intuition about programming, in the sense that no feasible implementation can produce an unbounded number.

Unbounded nondeterminism has been reconsidered by Park[7], who observed that the above contradiction could be explained on the grounds of the existence of two interpretations of nondeterminism: loose nondeterminism and tight nondeterminism. In tight nondeterminism, nondeterminism in the language describes more than one result, but a possible implementation does not have to produce all the results; it must merely guarantee that the results produced are described by the semantics. In loose nondeterminism, 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 nondeterminism of a scheduler of concurrent operations is a loose one.

Unbounded nondeterminism arises in the context of infinitely branching transition systems[44]; infinitely branching systems have been considered as a model for TCSP[36].

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, consider linear temporal logic[1,8,45], which is interpreted over (finite or infinite) sequences of program states. The exposition here is based on Pneuli[8]. A state formula is any well formed first-order formula; its truth value at instant i in a sequence $s = s_1 s_2 \cdots s_i \cdots$ is found by evaluating it on s_i. A temporal formula is constructed from state formulas p, q, to which the following (basic) operators are applied, and interpreted in state s_i of some sequence as follows:

- X: strong next instant operator (Xp is **true** at i if S_{i+1} exists and p is a state formula **true** at $i + 1$)

- U: *until* operator (pUq is **true** at i if there exists j 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 = $\neg$F$\neg$p), often denoted as box

Examples of temporal formulas are:

- p → Fq: p now implies eventually q
- G(p → Fq): every p is followed by a q
- F(Gp): eventually permanently p
- G(Fp): infinitely often p (for infinite sequences only)

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 $j < \text{len}(s)$, $s_j \models p$.

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

Temporal logic distinguishes between safety and liveness properties. A safety property is characterized as

Gp

where p is some past formula, that is, p holds over all finite prefixes. Any formula constructed out of past formulas, the logical operators $\wedge$ and $\vee$, and the future temporal operators G and 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, G(Fp), F(Gp)

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 → Fq, i.e., F($\neg$p $\vee$ q) = weak fairness
- G(Fp) → Fq, i.e., F(G($\neg$p) $\vee$ q) = strong fairness

Other properties useful to express scheduling constraints include (consider p, p_i as 'request', and q, q_i as 'response'):

- Pr(p,q) ≡ (Fq → ($\neg$qUp)) = p precedes q
- Pr(p_1,p_2) → Pr(q_1,q_2) = if p_1 precedes p_2 then q_1 precedes q_2 (FIFO)

This has given an overview of 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 power to 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[1]; for example, the fairness and responsiveness properties shown above could not have been expressed without U. Another point is that linear temporal logic, being a noncounting formalism[8] 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 all 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, has been introduced[46], thus making it possible to obtain the results of such a comparison. It is shown that CTL* strictly subsumes B(L(F,X,U)). The expressive power of branching-time logic B(L(F)) is different from that of its linear counterpart L(F).

Automata and infinitary languages

For fairness properties to be included in an 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 reconsidering finite state automata as acceptors of languages.

Infinitary languages have been investigated by a number of authors[28,47]. ω-regular languages are a natural extension of regular languages with ω-iteration operator, that is, iteration an infinite number of times. ω-automata[28] are formed from standard finite automata by the addition of a structure for accepting infinite sequences. Two (equivalent in the nondeterministic case) varieties are B-automaton, due to Büchi, with an additional set of 'green' states that must be visited infinitely often, and M-automaton, 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 of ω-regular languages and a concurrency operator (fairmerge) is dealt with by Park[28]. 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 noncommunicating concurrency, but it may be extended to a communication merge[31,48].

Fairness in (labelled) finite state automata has been discussed, considering only the case of strong fairness[13]. 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 that are fair with respect to a given notion is discussed. The relationship with Büchi and Muller automata is also considered.

Infinitary languages have been generalized to allow for concurrent behaviours[9]. Infinitary trace languages are developed, and a topological characterization of behavioural properties that includes fairness properties is presented. The notions of process and event fairness are distinguished, but the formalism is general enough to express notions such as equifairness and fair reachability from states.

Process algebras

Process algebras are algebraic languages for the specification of concurrent processes and the formulation of the properties of such processes. They are usually introduced together with the rules of algebraic calculus. Process algebras should not be treated as models for concurrency but as abstract representations of classes of such models.

CCS[39] 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), nondeterminism (+), concurrency (|), recursion (fix), and restriction (\). Synchronization of actions $a \in \Lambda$ with co-actions $\bar{a} \in \bar{\Lambda}$ is enforced only under restriction; otherwise processes may choose to proceed autonomously or synchronize. The following is an example of a CCS process:

$$(E|F)\backslash b$$

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

Fairness in (pure) CCS without restriction has been examined[22] and viewed 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 synchronize with other processes. Pure CCS with restriction has been investigated[6], 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 Costa and Stirling[6] 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 that identify the path to a component of an expression. These labels have then been used to identify uniquely autonomous actions and processes in a given expression. A subclass of live processes, that is, the set of processes that became active and have not engaged in a transition 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 cannot be similarly characterized and involves predictive choice[39], that is, the definition needs to be based on an infinite tail of the derivation in question.

A metric characterization of weak and strong fairness in CCS has been presented[67]. 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 composed of expressions and their derivations. Fairmerge has been discussed in a CCS setting[30].

CSP[36,49] (also known as TCSP) is a process algebra that distinguishes between internal and external choice and allows hiding (abstraction). Interprocess communication is based on (n-communicating) joint actions rather than on binary synchronization 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[49]. Extending the failures semantics has been proposed[44] so that unbounded nondeterminism is allowed. It is not known if fairness issues have been considered in this setting.

ACP[48], the Algebra of Communicating Processes, is not tied to a particular model. It is based on bisimulation semantics[28]. 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 has been extended with abstraction and encapsulation operators[50]. A process is fair if for any improbable path (i.e., one that contains infinitely many exits) the probability that it will be executed is zero. Fair abstraction rule, which 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 process algebra has been discussed[31]. 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 $A\langle\langle L\rangle\rangle$, where A is an agent is the sense of CCS and L is an ω-regular language that contains no finite sequences. $A\langle\langle L\rangle\rangle$ behaves as the agent L, except that its infinite behaviours must be members of L. Equivalences can then 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 revolves around 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[51] 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 $M: S \rightarrow Nat$ 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: $M_0 t_1 M_1 t_2 \ldots$. Petri nets can be viewed as transition systems, with markings forming the states.

Transition fairness has been investigated[33, 34] and an infinite hierarchy of notions of fairness, motivated by the need to exclude conspiracy, introduced. This hierarchy collapses to a single notion for a simpler class of nets (confusion-free). Another approach[11] defines notions of fairness for three granularity levels 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 have been investigated[11] and some decidability questions answered[52].

It is not clear how processes should be defined for Petri nets; a variety of decompositions are possible. A notion of a process based on occurrence nets has been introduced, and transition fairness and marking fairness discussed[12]; these notions are shown to be independent. The hierarchy of marking fairness is proved to collapse.

Livelocks have been investigated in parallel programs modelled by Petri nets and a technique for verifying the absence of livelocks introduced[42].

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 nondeterministic 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.

Labelled transition systems have been used as models for nondeterministic programs[3]. 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.

Transition fairness[3] corresponds to fairness of choice. When concurrency is represented as nondeterministic interleaving, however, transition systems may be used to model parallel composition of processes[45]. 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). Here (weak, strong, or unconditional) transition fairness corresponds to (weak, strong, or unconditional) process fairness. Transition systems have been extended to fair transition systems by adding restrictions on justice and fairness (i.e., weak and strong fairness)[8].

Fairness for a concurrent while language modelled as a transition systems has also been examined[19].

Asynchronous transition systems, namely, extensions of labelled transition systems with an independency relation, have been considered[9]. A notion of process structure is introduced; a variety of process structures for a given system are possible. Process structures give rise to a lattice of process fairness properties.

Denotational semantics

The issue of fairness raises a few important questions in denotational semantics[53], 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[19] is needed to provide denotations of nondeterministic or concurrent programs.

The notion of continuity, in the sense of preserving least upper bounds of infinite chains, has been extended onto uncountable chains[7]. A relational approach to denotational semantics of nondeterministic programs is proposed. The problem of (concurrency) fairness and unbounded nondeterminism is reconsidered, and 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. The latter suggests that generalized fixpoints may be needed to deal with fairness[7].

A powerdomain semantics for a concurrent while language (without internal nondeterminism) has been discussed[19]. Weak and strong event fairness are distinguished, and fair parallel operators $\|_n$ and $\|^n$, which put an explicit bound 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) for a process algebra has been addressed[54]. 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 sequences

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.

PROVING PROPERTIES UNDER FAIRNESS CONSTRAINTS

Properties of programs can be split into two classes: safety and liveness properties. This is motivated by the different proof techniques required in each case: for safety, structural induction based on invariants suffices, while liveness is based on the method of well founded sets. A number of axiomatic systems that originate from the Floyd assertion method have been introduced to reason about the correctness of concurrent programs[8,41,45,55]. Each axiom corresponds to a syntactic 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 liveness properties) of programs is based on well founded sets. A well founded set (A, <) is a partially ordered set in which no infinite strictly decreasing sequence exists. Elements of such a set may be used to define a ranking function[2] $\rho: S \rightarrow A$ [2], 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. As no infinite ranking sequence is possible, the program must terminate. The program terminates if, and only if, such a ranking function exists[2].

For deterministic programs, it is sufficient to require that the ranking function decreases at every step. When concurrent, or nondeterministic, programs are considered, however, this requirement may be too strong. Often, infinite sequences are not admissible under a fairness constraint, which suggests that, in many cases, the ranking function could be defined only for the fair sequences. Consider 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. If (weak process) fairness is assumed, however, the program terminates.

The solution suggested[2] is to define ranking sequences, which only decrease eventually. Thus distinguish between a helpful direction, i.e., one that decreases the ranking function ('b := **false**' in the example above), and an indifferent one, i.e., one that does not increase it. Now, using a proof rule that incorporates the required fairness constraint, it can be shown 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 nondeterministic 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: $0 = \varnothing$, $n + 1 = n \cup \{n\}$. Now proceed by taking $\omega = \{0, 1, \ldots\}$, $\omega + 1 = \omega \cup \{\omega\}$, etc. Ordinals are strictly ordered by the set membership and form a well founded class.

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 defined on the states identified with the contents of the variable comm is: ρ(**true**) = ω, ρ(**false**) = x; comm has been 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 other liveness properties[8], not just termination. In this approach, fairness is incorporated as a constraint within a proof rule; a variety of proof rules can be formulated, depending on the actual notion of fairness.

A compositional approach is also possible, which allows for modules to be specified and verified, independent of their environment. This is achieved by the introduction of a notion of an interface. A compositional approach to temporal logic has been discussed[56], but it is not clear how fairness constraints are dealt with.

Well founded sets are also used to prove termination of nondeterministic programs[5]. Two inter-reducible meth-

ods 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 together with their soundness and semantic completeness[5]. Termination proofs are considered sufficient since other liveness properties can be reduced to termination[57].

Transformational approach

An alternative method for tackling termination of nondeterministic programs is the method of explicit scheduler[5]. 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 that decreases at every computation step. Thus a simpler rule can 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 has been analysed, where soundness and completeness are also considered[5].

Positive approach

The methods 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[6]. 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[27] 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[27], 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 that the number of states grows exponentially.

Other model checking systems that allow for fairness have been reported[58,59].

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 synchronization or data transfer over an unreliable medium.

Concurrent algorithms

When designing and implementing mutual-exclusion algorithms, some reasoning about fairness is needed to ensure that a process that requests access to the critical section will eventually be allowed to enter. Mutual-exclusion algorithms are often verified to exemplify the proof technique under consideration. Peterson's algorithm has been shown to be correct (the proof is based on linear temporal logic)[8,45]. Mutual exclusion using the 'proof lattice' technique has been proved[26]. Peterson's algorithm has been automatically verified by the model checking algorithm for the CTL temporal logic[27]. A version of n-process mutual exclusion has been verified using the notion of extreme fairness (that is, probabilistic fairness)[4]; the algorithm is based on symmetric protocols that do not share a writable 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, have been analysed[60]. Apart from fairness, some fault-tolerance properties are also considered there.

Fairness of synchronization primitives has been characterized[61]. Unbounded communication primitives and two versions of semaphore operations are axiomatized. Monitors that guarantee fairness have been implemented as an extension of PASCAL with concurrency[62].

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[31] and an automatic verification in

CTL[27]. The CSMA/CD protocol has also been verified using an infinitary process algebra[31].

Verification of multiprocess probabilistic protocols has been discussed[4], where the notion of extreme fairness is employed. The protocol used there is based on an n-process mutual-exclusion algorithm that assumes no writable storage for communication.

Automatic verification using a model checking system written in PROLOG has also been discussed[58]. More detailed information on the subject of protocol verification is given elsewhere[31].

Programming languages

It is an interesting question whether fairness constraints formulated, at the stage of the language design or implementation, for existing programming languages are adequate. Criteria for appraisal of fairness notions have been introduced[63]. These are:

- feasibility, that is, whether some computation of a program remains once all computations that are unfair with respect to the given notion have been excluded
- equivalence robustness, that is, if fairness respects the equivalence induced by the model
- liveness enhancement, which requires that additional liveness properties hold for some program.

When fairness notions for CSP[15] have been analysed, only strong process fairness (see the section 'Taxonomy of fairness notions') is shown to satisfy all three criteria. Several fairness notions for communication in ADA have also been considered.

Other work concerning programming languages revolves around synchronization primitives, for example, semaphores[61] or monitors[62].

CONCLUSIONS

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 usually relating to the lack of progress of some component of a system, must be disallowed. When first introduced[2,16,28,41], 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 multiprocessor 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 dependent on 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, and progress has been made need to be formalized 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[8], 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[9,31] 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[27].

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[1,46]. In verification techniques, incorporating fairness has called for transfinite induction[2,5]. In denotational semantics, continuity had to be reconsidered[27].

Further difficulty arises when different behavioural structures are used as models, for example, when nondeterministic interleaving is used to represent concurrency. At this high level of abstraction, no restrictions are usually imposed on admissible sequentializations of concurrent computations (such as the finite delay property[16]). 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[9,64].

Owing to the 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 as to whether powerful formalisms like transfinite induction are really needed to deal with a property so easy, on the face of it, to implement in practice. Also, some quantitative methods, rather than the qualitative ones that have been applied until now, would be beneficial to model the bounded delays necessary in realtime systems.

Recently, some criticism was raised[65] as to whether

fairness is a 'workable notion' as no finite experiment can be set up to prove or disprove it. The author would argue with this statement and tend towards thinking that fairness is a useful abstraction[66], especially when related to concurrency, and, as such, although undetectable by finite experiments, it is nevertheless a property that can be formally stated[9]. Further research is needed, however, to provide a comparative study 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.

REFERENCES

1 **Gabbay, D, Pnueli, A, Shelah, S and Stavi, J** 'On the temporal analysis of fairness' in *Proc. 7th ACM Symp. Principles of Programming Languages* (1980)

2 **Lehman, D, Pnueli, A and Stavi, J** 'Impartiality, justice and fairness: the ethics of concurrent termination' in **Even, S and Kariv, O (eds)** *Proc. Automata, Languages and Programming.* (Lecture Notes in Computer Science Vol 115) Springer-Verlag, Berlin, FRG (1981)

3 **Queille, J P and Sifakis, J** 'Fairness and related properties in transition systems-a temporal logic to deal with fairness' *Acta Inf.* Vol 19 (1983) pp 195-220

4 **Pnueli, A and Zuck, L** 'Verification of multi-process probabilistic protocols' *Distrib. Comput.* Vol 1 (1986) pp 53-72

5 **Francez, N** *Fairness* Springer-Verlag, New York, NY, USA (1986)

6 **Costa, G and Stirling, C** 'Weak and strong fairness in CCS' *Inf. Computation* Vol 73 (1987) pp 207-244

7 **Park, D** 'On the semantics of fair parallelism' in **Bjorner, D (ed)** *Abstract Software Specifications, Proc.* (Lecture Notes in Computer Science Vol 86) Springer-Verlag, Berlin, FRG (1980)

8 **Pnueli, A** 'Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends' in **de Bakker, J W, de Roever, W P and Rozenberg, G (eds)** *Current trends in concurrency* (Lecture Notes in Computer Science Vol 224) Springer-Verlag, Berlin, FRG (1986)

9 **Kwiatkowska, M Z** 'Fairness for non-interleaving concurrency' *PhD Thesis* University of Leicester, Leicester, UK (1989)

10 **Kuiper, R and de Roever, W P** 'Fairness assumptions for CSP in a temporal logic framework' in **Bjorner, D (ed)** *Proc. TC.2 Working Conf. Description of Programming Concepts* North-Holland, Amsterdam, The Netherlands (1983)

11 **Carstensen, H and Valk, R** 'Infinite behaviour and fairness in Petri nets' in **Rozenberg, G (ed)** *Advances in Petri Nets 84* (Lecture Notes in Computer Science Vol 188) Springer-Verlag, Berlin, FRG (1984)

12 **Merceron, A** 'Fair processes' in *Proc. 7th European Workshop on Applications and Theory of Petri Nets* (Oxford, UK, 1986)

13 **Priese, L, Rehrmann, R and Willecke-Klemme, U** 'An introduction to the regular theory of fairness' *Theoret. Comput. Sci.* Vol 54 (1987) pp 139-163

14 **Kwiatkowska, M Z** 'A survey of fairness notions' *Technical Report No 14* University of Leicester, Leicester, UK (1988)

15 **Hoare, C A R** 'Communicating sequential processes' *Commun. ACM* Vol 21 No 9 (1978) pp 666-677

16 **Karp, R M and Miller, R E** 'Parallel program schemata' *J. Comput. Syst. Sci.* Vol 3 (1969) pp 147-195

17 **Shields, M W** 'Behavioural presentations' in **de Bakker, J W, de Roever, W-P and Rozenberg, G (eds)** *Linear time, branching time and partial order in logics and models for concurrency* (Lectures Notes in Computer Science Vol 354) Springer-Verlag, Berlin, FRG (1989)

18 **Shields, M W** *Elements of a theory of parallelism* to be published

19 **Plotkin, G** 'A powerdomain for countable non-determinism' in *Proc. Automata Languages and Programming, 9th Coll* (Lecture Notes in Computer Science Vol 140) Springer-Verlag, Berlin, FRG (1982)

20 **Grumberg, O, Francez, N and Katz, S** 'A complete rule for equifair termination' *J. Comput. Syst. Sci.* Vol 33 (1986) pp 313-332

21 **Mazurkiewicz, A, Ochmanski, E and Penczek, W** 'Concurrent systems and inevitability' *Theoret. Comput. Sci.* Vol 64 (1989) pp 281-304

22 **Costa, G and Stirling, C** 'A fair calculus of communicating systems' *Acta Informatica* Vol 21 (1984) pp 417-441

23 **Lamport, L** 'Proving the correctness of multiprocess programs' *IEEE Trans. Soft. Eng.* Vol 3 No 2 (1977) pp 125-143

24 **Apt, K R and Olderog, E-R** 'Transformations realizing fairness assumptions for parallel programs' in *Proc. Symp. Theoretical Aspects of Computer Science* (Lecture Notes in Computer Science Vol 166) Springer-Verlag, Berlin, FRG (1984)

25 **Alpern, B and Schneider F B** 'Defining liveness' *Inf. Process. Lett.* Vol 21 (1985) pp 181-185

26 **Owicki, S and Lamport, L** 'Proving liveness properties of concurrent programs' *ACM Trans. Programm. Lang. Syst.* Vol 4 No 3 (1982) pp 455-495

27 **Clarke, E M and Grumberg, O** 'Research on automatic verification of finite-state concurrent systems' *Ann. Rev. Comput. Sci* Vol 2 (1987) pp 269-290

28 **Park, D** 'Concurrency and automata on infinite sequences' in **Deussen, P (ed)** *Proc. 5th GI Conf. Theoretical Computer Science* (Lecture Notes in Computer Science Vol 104) Springer-Verlag, Berlin, FRG (1981)

29 **Apt, K R and Francez, N** 'Modelling the distributed termination convention in CSP' *ACM Trans. Program. Lang. Syst.* Vol 6 No 3 (1984) pp 370-379

30 **Hennessy, M** 'An algebraic theory of fair asynchronous communicating processes' *Theoret. Comput. Sci.* Vol 49 (1987) pp 121-143

31 **Parrow, J** 'Fairness properties in process algebra with applications in communication protocol verification' *PhD Thesis* Uppsala University, Sweden (1985)

32 **Pnueli, A** 'On the extremely fair treatment of probabilistic algorithms' in *Proc. 15th ACM Symp. Theory of Computing* (1983) pp 278–290

33 **Best, E** 'Fairness and conspiracies' *Inf. Process. Lett.* Vol 18 (1984) pp 215–220

34 **Best, E** 'Erratum' *Inf. Process. Lett.* Vol 19 (1984) p 162

35 **Attie, P, Francez, N and Grumberg, O** 'Fairness and hyperfairness in multi-party interactions' submitted to *Distributed Comput.*

36 **Brookes, S D, Hoare, C A R and Roscoe W** 'A theory of communicating sequential processes' *J. ACM* Vol 31 No 3 (1984) pp 560–599

37 **Lauer, P E, Shields, M W and Best, E** 'Design and analysis of highly parallel and distributed systems' in **Bjorner, D (ed)** *Abstract Software Specifications Proc.* (Lecture Notes in Computer Science Vol 86) Springer-Verlag, Berlin, FRG (1980)

38 **Kwiatkowska, M Z** 'Modelling concurrency using ambiguous asynchronous transition systems' *Technical Report No 9* University of Leicester, Leicester, UK (1988)

39 **Milner, R** *A calculus for communicating system* (Lecture Notes in Computer Science Vol 92) Springer-Verlag, Berlin, FRG (1980)

40 **Hennessy, M** 'Modelling finite delay operators' *Technical Report CSR-153-83* University of Edinburgh, Edinburgh, UK (1983)

41 **Ashcroft, E A** 'Proving assertions about parallel programs' *J. Comput. Syst. Sci.* Vol 10 (1975) pp 110–135

42 **Kwong, Y S** 'On the absence of livelocks in parallel programs' in **Kahn, G (ed)** *Semantics of Concurrent Computation Proc.* (Lecture Notes in Computer Science Vol 70) Springer-Verlag, Berlin, FRG (1979)

43 **Dijkstra, E W** *A discipline of programming* Prentice-Hall, Englewood Cliffs, NJ, USA (1976)

44 **Roscoe, A W** 'Two papers on CSP' *Technical monograph PRG-67* University of Oxford, UK (1988)

45 **Manna, Z and Pnueli, A** 'Temporal verification of concurrent programs' in **Boyer, R S and Strother Moore, J (eds)** *The correctness problem in computer science* Academic Press, New York, NY, USA (1981) pp 215–273

46 **Emerson, E A and Halpern, J Y** 'Sometimes' and 'not never' revisited: on branching versus linear time temporal logic' *J. ACM* Vol 33 No 1 (1986) pp 151–178

47 **Boasson, L and Nivat, M** 'Adherences of languages' *J. Comput. Syst. Sci.* Vol 20 (1980) pp 285–309

48 **Bergstra, J A and Klop, J W** 'Algebra of communicating processes' in **de Bakker, J W, Hazenwinkel, M and Lenstra, J K (eds)** *Proc. CWI Symp. Math. & Comp. Sci.* Amsterdam, The Netherlands (1986)

49 **Hoare, C A R** *Communicating sequential processes* Prentice-Hall, Englewood Cliffs, NJ, USA (1984)

50 **van Glaabbek, R J** 'Bounded non-determinism and induction principle in process algebra' *Technical Report CS-R8634* Centrum foor Wiskunde en Informatica, Amsterdam, The Netherlands (1986)

51 **Rozenberg, C and Thiagarajan, P S** 'Petri nets: basic notions, structure, behaviour' in **de Bakker, J W de Roever, W P and Rozenberg G (eds)** *Current trends in concurrency* (Lecture Notes in Computer Science Vol 224) Springer-Verlag, Berlin FRG (1986)

52 **Carstensen, H** in **Brandenburg, F J, Vidal-Naquet, G and Wirsing, M (eds)** *Proc. STACS 87* (Lecture Notes in Computer Science Vol 247) Springer-Verlag, Berlin, FRG (1987)

53 **Stoy, J E** *Denotational semantics: the Scott-Strachey approach to programming language theory* MIT Press, Cambridge, MA, USA (1977)

54 **de Bakker, J W, Meyer, J-J C and Olderog, E-R** 'Infinite streams and finite observations in the semantics of uniform concurrency' *Theoret. Comput. Sci.* Vol 49 (1987) pp 87–112

55 **Owicki, S and Gries, D** 'An axiomatic proof technique for parallel programs' *Acta Inf.* Vol 6 (1976) pp 319–340

56 **Barringer, H, Kuiper, R and Pnueli, A** 'Now you may compose temporal logic specifications' in *Proc. 16th Symp. Theory of Computing* (1984) pp 51–63

57 **Grumberg, O, Francez, N, Makowsky, J A and de Roever, W-P** 'A proof rule for fair termination of guarded commands' *Inf. Contr.* Vol 66 (1985) pp 83–102

58 **Cavalli, A R and Horn, F** 'Proof of specification properties by using finite state machines and temporal logic' in *Proc. IFIP WG 6.1 7th Int. Conf.* (1987)

59 **Emerson, E A and Lei, C-L** 'Modalites for model checking: branching time logic strikes back' *Sci. Comput. Program.* Vol 8 No 3 (1987) pp 275–306

60 **Lamport, L** 'The mutual exclusion problem. II. Statement and solutions' *J. ACM* Vol 33 (1986) pp 327–348

61 **Martin, A J** 'An axiomatic definition of synchronisation primitives' *Acta Informatica* Vol 16 No 2 (1981) pp 219–235

62 **Karp, R A and Luckham, D C** 'Verification of fairness in an implementation of monitors' in *2nd Int Conf. Software Engineering* IEEE, Los Alamos, CA, USA (1976)

63 **Apt, K R, Francez, N and Katz, S** 'Appraising fairness in languages for distributed programming' in *Proc. 14th ACM Symp. Principles of Programming Languages* (1987)

64 **Kwiatkowska, M Z** 'Event fairness in asynchronous transition systems' *Technical Report No 10* University of Leicester, Leicester, UK (1988)

65 **Dijkstra, E W** 'Position paper on fairness' *Soft. Eng. Notes* Vol 13 No 3 (April 1988) pp 18–20

66 **Chandy, K M and Misra, J** 'Another view of fairness' *Soft. Eng. Notes* Vol 13 No 3 (July 1988) p 20

67 **Costa, G** 'A metric characterization of fair computations in CCS' *Internal report CSR-169–84* University of Edinburgh, UK (1984)