Appendix 1

Describing concurrent behaviour

This appendix contains a description of current techniques for describing concurrent behaviour.

Appendix 1 Describing concurrent behaviour

The purpose of this appendix is to summarise existing work, without attempting to explicitly relate it to the major themes of the thesis. Information has been drawn from the work of many authors, including [Pnueli 86a] [Pnueli 86b] [Emerson 88] [Reisig 88] [Gaifman 88]. I would also like to thank the following for helpful private communication: J. Buntrock, S. Katz, B. Monahon

We focus principally on systems of concurrent procedural processes, and consider two major concerns: how to represent a *run* of the system and how to group runs together so as to describe the abstract behaviour of the system. A run is a representation of a single execution of the system, typically constructed by interleaving (i.e. sequentialising) the *events* from the participating processes to produce a *trace*. In this context an event is a single instruction from a process, which can be regarded as atomic with respect to the other instructions in the processes. At least two formalisms are available for traces of runs of a system: *state-based* and *event-based* views. These respectively focus on the states of the system and the transitions between states.

In the state-based view states are the principal observable entities in the behaviour of the system, with transitions (i.e. events) serving only to explain changes to the current state. Programs using this method describe the sequence of states through which the program passes. State-based traces typically are most useful for shared-variable models of concurrency, where the new states caused by executing instructions are of principal importance.

In the event-based approach, transitions are the important entities by which the behaviour of the system is described, and states are seen as purely a by-product of transitions. This view is of use in message-passing systems, where the principal events are the sending and receiving of messages along channels. Programs using this method consists of an interleaving of events which are observed in the system.

Whilst the distinction between the two approaches may seem unimportant, it has ramifications for the second question, namely how to group runs of a system together in such a way as to be able to both uniquely and abstractly describe the behaviour of the system. There are three major suggestions proposed, which are known as the linear, branching and partial order behaviour structures.

The linear approach consists of grouping all the possible traces of the system into a set, and defining the behaviour of the system by examining properties of sequences in this set. Such an approach requires predicates on sequences (i.e. over all future traces), and cannot consider questions such as "process *p* may do action *x*".

The branching approach groups traces into computation trees, and requires a specification language which can predicate on the structure of this tree. This approach considers the set of *possible* future traces, which means that it has more expressive power than the linear approach, being able to formulate assertions such as "process *p* may do action *x*".

Partial orders considers relations between structures of events. This differs from the previous two approaches in that now consideration is only made of events which are in some sense causally dependent on other events. This dependence relation § may be such that $e_1 § e_2$ iff e_1 must temporally precede e_2 (the causality relation, e.g. event structures [Winskel 82] [Winskel 88]), or that e_1 cannot occur in the same run as e_2 (the exclusion relation). The set of all possible traces can be constructed by interleaving all permutations of independent events with those sets which respect the causal dependencies.

Partial orders are more activity-oriented than the other two approaches (which are event-oriented), and any ordering which can affect the outcome (i.e. not just the behaviour) of the program should be represented as a causal dependence in the partial order. As an example, one can view a variable as a process which receives signals (e.g. evaluate, write), to which it must respond. If two signals arrive simultaneously, then they are arbitrarily sequentialised. In particular, partial orders do not require an interleaving partial semantics.

The methods proposed for the representation of the behaviour of a concurrent system can be partitioned into interleaving and non-interleaving approaches [Kwiatkowska 88]. An interleaving approach represents the behaviour of the system by an interleaving of events, and so cannot semantically distinguish concurrent processes from those which are sequential but non-deterministic (i.e. those which make arbitrary choices as to which action to perform next). Examples of interleaving systems include CSP ([Hoare 78] [Hoare 85], [Brookes et al 84]), CCS ([Milner 80],[Milner 83]) and temporal logic ([Ben Ari et al 83],[Manna & Pnueli 81]).

Non-interleaving approaches, on the other hand, do not represent execution of a concurrent system by means of an interleaving, but (typically) in the form of a a partial ordering, where events are ordered only when they are causally dependent on each other. Any permutation of unordered events yields a total ordering, i.e. a possible interleaving of events. Non-interleaving approaches can distinguish between concurrency and non-determinism. Petri Nets ([Petri 62] [Reisig 85] [Rozenberg & Thiagarajan 85]), event structures ([Winskel 82] [Winskel 88]) and behavioural presentations ([Shields 88]) utilise this approach.

All of these approaches have been the subject of a large amount of research, and their semantics are well understood. For example, CSP provides a mechanism for modularity with its hiding operator, and statecharts permit the explicit representation of concurrency. Axiomatisations for some theories of concurrency have been found, such as [Brookes 83] and [deNicola 83] for CSP and [van Glabbeek 86] and [Milner 80] for CCS.

All approaches make use of certain assumptions (i.e. fictions or abstractions) about systems:

- as already stated, events are instantaneous actions which have no duration in time. If this assumption is not made, then consideration must be made of possible interference between two processes acting asynchronously. This appears to be no more than a convenient simplification for the full-scale notion of atomicity: events are atomic if no other interfering action can occur between the time the event starts and the time the event ends. Any independent events which occur concurrently can be sequentialised arbitrarily to give an interleaving. The time that the event takes is irrespective of its atomicity.

- interleaving is a fair representation of concurrency. This is not in general true: consider the program {y=0} y++ || y--. If all events are interleaved (or take zero time - essentially the same assumption that actions are atomic) then the result will always be {y=0}. In a real concurrent system, due to the lack of atomicity of assignments, the answer could be anything in the set {-1,0,1}. However if we somehow restrict the possible events which can occur in one transition concurrently to those which only involve one *critical event*, then the assumption holds. For example, in a shared variable system a critical event is a *critical reference*. A critical reference in a shared variable system is either a writing reference to a variable which can be read or written to by a parallel statement, or a reading reference to a variable which can be written to by a parallel statement.

- a global state. This assumption is necessary to give some meaning to an assertion such as {x+y>0}. One might decide that the assertion can still be tested in a concurrent system lacking the instantaneous actions idealisation, in that it is tested (in the above example) whenever neither x nor y are being written to. A non-trivial assumption, since e.g. should a program x++||x-- with the global assertion {x=0} hold with x initialised to 0?

Discriminatory power is an important attribute of a (system or requirements) specification methodology, so we shall now illustrate the difference between the three behavioural representations. Consider four programs specified in the CCS notation[1], each of which assumes that events are instantaneous actions having zero duration:

(i) $a(b+c)$

event a occurs, at which point a decision is made as to whether to do event b or c

(ii) $ab+ac$

a decision is made as to whether to perform event a followed by b, or a followed by b

(iii) $a \,||\, b$

a is performed in parallel with b

(iv) $ab + ba$

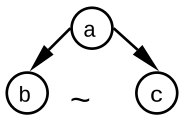
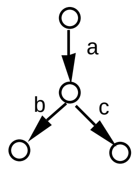
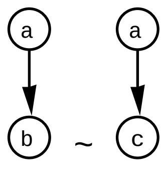
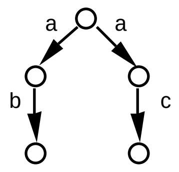
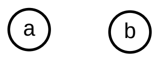
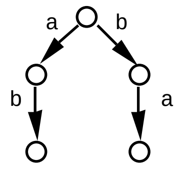
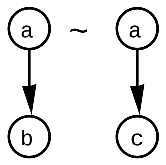
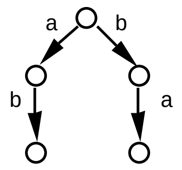
a non-deterministic choice is made as to whether to execute a then b or b then a

These examples are used in Figure A1 (taken from [Pnueli 86a]) to illustrate the differences between the three approaches, by considering the representation of the four programs using the three approaches. As can be seen, the linear approach cannot discriminate between **(i)** and **(ii)** - a (non-deterministic) choice being made before or after event a. Both the branching time and partial order approaches differentiate between splitting and non-determinism. However, only a partial order can differentiate between concurrency and non-determinism, as exemplified by programs **(iii)** and **(iv)** respectively. Such a discriminatory power is perhaps unjustified, since we are assuming that all events take zero time, and therefore concurrency is equivalent to non-determinism (in that events must occur at distinct points in time, so they can be represented by a non-deterministic interleaving). Due to the expressive power of partial orders, they prevent implementation of a concurrent program by multiprogramming, and also do not

[1] these are not meant to be interpreted as CCS programs, since in CCS there is no semantic difference between concurrency (a||b) and non-determinism (ab+ba). Such an equivalence may not be the desired interpretation, since collapsing concurrency into interleavings may not be suitable for building some sorts of concurrent systems, as it precludes some (otherwise possible) interleavings.

admit of the (direct) parallelisation of a sequential algorithm.

As can be seen, linear, branching and partial order proceed in order from less discriminating to more discriminating, and from more abstract to less abstract. It therefore seems sensible to suggest that linear approaches are more useful at the requirements specification stage, where abstractness is desirable, and partial order approaches are more useful at the system specification stage, where more discriminatory power is appropriate.

Approach / Behaviour	Partial Order Event Structure	Branching Tree	Linear Set
a (b + c)			{ ab , ac }
ab + ac			{ ab , ac }
a || b			{ ab , ba }
ab + ba			{ ab , ba }

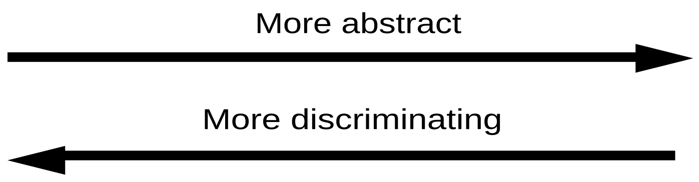


Figure A1