Normal forms for synchronous and asynchronous process calculi.

Discussion paper between Tony Hoare and Cedric Fournet and Silvano dal Ziglio.

~~Revised 1999-12-09, and circulated for comment.~~

Revised March 2000, and circulated again.

And again, at the end of March.

Summary

Equivalences in process algebra have long been the subject of successful research, in which bisimulation plays an important role. The next stage in the progress of mathematical ~~research~~ is to choose canonical representatives in each equivalence class, and use them to simplify subsequent calculations. Ideally, the choice of bisimulation should be made on semantic or pragmatic grounds, to reflect the purpose and application of the calculus. The canonical closure condition which selects the normal form is derived from the bisimulation. The definition of the operators of the calculus may need to be weakened (as little as necessary) to ensure that the bisimulation is a congruence; weakening preserves the correctness of the operational semantics, but usually at the expense of full abstraction. The hope is that the normal form, and all theorems relating to it, can be shared by many different calculi and different bisimulations.

This note is issued as a challenge for future research. I would like to know whether anybody in the world would be interested in using the results of the research, if it is successful. Next, would anybody actually like to do the research?

Background

Symbolic calculation is a useful skill for engineers engaged in the design and validation of advanced technological products. The steps of the calculation are justified by algebraic laws, proved by mathematicians within the relevant scientific theory. A calculus is called complete if there are enough laws to prove every equation that is true in the theory. Preferably, the proofs should be automated, for example by reduction of both sides of an equation to some kind of normal form. Inequality of normal forms should be easily detectable, preferably by display of a counter-example.

Process algebra is the branch of mathematics that has been developed to analyse, predict, and control the behaviour of networks of interacting computers and programs. It deals with issues of security, robustness and performance. Its great achievement has been to apply the same mathematical abstractions at all levels of scale, from instruction pipelines in a single microprocessor through multiprogrammed personal computers, clusters and networks, right up to the entire world wide web. At the lower end of the scale, communication is usually synchronised, with input of a message occurring instantaneously with its output. Any overhead involved is minimal. On a larger scale, there is an inescapable delay between output and input of a message, and overheads can be significant. Different branches of process algebra have been developed to deal with the synchronous and asynchronous cases.

On occasion, there is a clear need for a calculus that combines both these modes of interaction. For example, asynchronous message passing is generally implemented on synchronous hardware; conversely, control signals, atomic actions, exceptions, snapshots and checkpoints have to be implemented on asynchronous networks as if they were instantaneous. Furthermore, we may want to take advantage of the characteristics of the architecture on which the programs are loaded by transforming programs and protocols between these two modes, using correctness-preserving transformations. It may be easier to reason about these issues if the relevant paradigms are embedded in the same calculus. Finally, we would like to show how commonality mathematical results can be transferred and shared between different process calculi.

We propose to base our work on some known normal forms for synchronous and asynchronous process calculi [Hennessy, de Nicola, Brookes, Roscoe, Bergstra, Mazurkiewicz, Baeten], and suggest how they may be combined. In this way we hope to contribute to the goal of unification pursued in the ESPRIT basic research action CONCUR.

Discussion.

The known disadvantages of normal forms are

1. They involve taking limits of an infinite series of approximations, e.g., the Taylor expansion of an infinitely differentiable function. However, many of the symbolic calculations can be performed on the finite approximations, with only a single inductive (or co-inductive) step at the end.

2. They may use notations outside the limits of the familiar language used to state the problem, e.g., complex exponents in the Fourier expansion of a real periodic function. However, such notations can suggest concepts that turn out to be independently interesting and useful for specification, calculation, and reasoning about designs.

3. They cause hideous expansion in the size of the formula, e.g., disjunctive normal forms in the propositional calculus. Sometimes the only contribution of the normal form is merely to guarantee the completeness of the calculus.

4. There is no unique or best choice of the shape of a normal form, e.g., polynomials can be expressed either as a sum of powers or a product of binomials. But this is actually an advantage, if each form is well adapted for a different purpose, and especially if conversions between them are practicable.

5. The uniqueness of a normal form often depends on some extra closure condition. For example a disjunctive normal form that is to be taken as canonical must have disjuncts that contain every one of the variables, either positively or negatively. Such closure conditions can be quite complex and non-intuitive.

There are three additional disadvantages in a project to find normal forms that are specific for process calculi.

6. There are so many different calculi that a normal form for a single calculus will have only a very small user community – and then only for a short time, until they move on to the next calculus. A solution to this problem is to design a normal form that can be applied to many different calculi. The differences between the calculi should be reflected in different closure conditions.

7. The choice of a normal form commits to a single notion of equality between processes, rather than the wide choice of equivalences offered by many different forms of bisimulation. Indeed, for practical application that may be an advantage. For purposes of research, some flexibility can be preserved, again by choice of alternative closure conditions.

8. The primitive operator of a process calculus is parallel composition. It is defined differently in each calculus by means of an operational semantics. But its simplest definition in terms of normal forms may involve a succession of operations, for example a product followed by restriction, hiding, and re-closure. For reasoning within a single simple calculus, induction on the structure of that particular syntax may be easier than equational reasoning based on the more general algebra. This may be the main reason why research into normal forms has been delayed, and perhaps even now is not worth the trouble.

Method

We start with the simplest kind of process calculus, one like SCCS, that is deterministic and synchronous. The dynamic behaviour of such a process is represented by its set of traces, i.e., the sequences of events in which it can engage. To represent a deterministic process, a set of sequences must satisfy a closure property: it is non-empty and prefix-closed. Because of this, the normal form can be drawn as an edge-labelled tree. The nodes represent states, with the initial state at the root. Each edge is labelled by the event that accompanies a transition from its source node to its destination. The traces of the process are just the sequences of events that label paths that start from the root of the tree. Because of determinism, the node at the end of each path is uniquely defined by the trace.

Let P be a process, and let s be a trace of its behaviour. We define P/s (P after s, the s-derivative of P) as the sub-tree of P that has as its root the node reached by the path s. Thus for example

$$t \text{ is in } P/s \quad \text{if and only if} \quad st \text{ is in } P$$

$$P/\langle\rangle = P \qquad P/st = (P/s)/t$$

If s is not a trace of P then P/s is the empty set (not a process). The inclusion of this mythical value (miracle) in the algebraic calculations is a matter of convenience. It will be inexpressible in any implementable calculus.

Now suppose $\equiv$ to be an equivalence relation between traces, which is of interest on pragmatic or semantic grounds. For example, we may wish to ignore the difference between two events a and b. We therefore regard two traces as equivalent if they differ only in the replacement of a by b or vice versa. Such an equivalence plays the role of a bisimulation in our investigations. But to justify this title, we require it to be respected by extension

$$s \equiv t \;\Rightarrow\; sv \equiv tv$$

A process P is said to respect this equivalence if its future behaviour never depends on which of two equivalent traces have occurred in the past

$$s \equiv t \;\Rightarrow\; P/s = P/t$$

An arbitrary process can easily be made to respect an equivalence by adding to it all traces equivalent to a trace that it already contains. We therefore define a closure operator

$$[P] = \{s \mid \text{there is a } t \text{ in } P \text{ satisfying } t \equiv s\}$$

For example, if $P = \{<>, <a>, <b>, <ac>, <bd>\}$, its closure also contains $<ad>$ and $<bc>$. Thus after occurrence of a (or b, because we cannot tell which), there seems to be a possibility of either c or d. But in the original process before closure, initial occurrence of a would have precluded the possibility of d. If d is what the user wants, there is a severe risk of deadlock. What is worse, the outcome is non-deterministic: if the first event was in fact b (and we can't tell), the deadlock will be avoided. The closure has failed to warn us of the risk.

We will postpone the proper treatment of non-determinism to the next section. Meanwhile, we will concentrate on a deterministic subset of a calculus like dataflow. Determinism is preserved by ensuring that whenever one member of an equivalence class of events can occur, then none of the other ones can.

The definition of bisimulation in terms of equivalence extends readily to a pre-ordering $\leq$ between traces

$$s \leq t \;\Rightarrow\; P/s \text{ includes } P/t$$

Such an ordering corresponds to a one-sided bisimulation, so we call it a simulation. It too must be respected by extension.

The parallel combination $P \times Q$ is a process that engages in a single instantaneous event (a,b) just when P engages in a and Q simultaneously engages in b. Its traces are just sequences of such synchronised events; the left projection is a trace of P and the right projection is a trace of Q

$$(P \times Q)/s = (P/\text{left } s) \times (Q/\text{right } s)$$

Consequently, if P and Q both respect an ordering, PxQ respects the product ordering. This definition is like that of a product in category theory. It is the only binary operator required in a process calculus, because all other kinds of binary operator can be obtained by applying unary operators to its result.

For example, there may be pairs of events *(a,b)* that cannot in practice ever occur together. These can be removed from the resulting process by restriction, as in CCS. Let *c!x* denote output of message value *x* on channel *c* and let *c?y* denote input of a message value *y* on the same channel. Now remove from the process *PxQ* all events *(c!x,c?y)* for which *x* is not equal to *y*. By ensuring that the inputting process Q allows all possible values of *y*, we achieve the effect of communication of value *x* from *P* to *Q* along channel *c*.

The above definition of parallel composition is not associative; but it can be made so by a standard trick. Define any event of the form *(a,(b,c))* as equivalent to *((a,b),c)*, and define two traces as equivalent if they differ only by substitution of equivalent events. Now add to the traces of *PxQ* all traces equivalent to any trace that it already contains. This closure under equivalence always produces a result that respects the equivalence, because all the unwanted distinctions are ignored.

A similar technique may be used to force symmetry. Unfortunately, in practice such a closure would introduce non-determinism, because the subsequent behaviour of a process may depend on whether the first event was an *(a,b)* or a *(b,a)*. This is why CSP restricts synchronised events to those of the form *(a,a)*; in other deterministic calculi like data flow, input and/or output on a given channel is allowed only to a single process. This ensures that the event *(a,b)* is possible only when *(b,a)* is not.

We now move on to a new kind of free-step calculus more like CCS, in that it relaxes the strict lock-step synchronisation of SCCS, by allowing events from one process to occur independently from the other. Let * denote a null event (hiaton), which occurs whenever a process has no interest in the event occurring in its environment. Thus in a trace of *PxQ*, an event *(a,*)* represents an action of *P* that is ignored by *Q*; *(*,b)* represents a similar lone action of *Q*; and *(a,b)* represents the simultaneous occurrence of both actions. *(*,*)* represents the hiaton of the parallel process. Variations on the basic definition of parallel composition may be made later by restriction. For example, lockstep synchronisation can be re-introduced into a free-step calculus by restricting all the lone actions. At the other extreme, in pure interleaving all synchronised events except *(*,*)* are restricted away. We would then like to equate the lone event *(*,a)* with *(a,*)*, but that would introduce non-determinism.

In order to support the concept of a hiaton as a null action, we require that a process always allows this event to occur, and that its behaviour is never affected by it. We therefore define the concept of a free-step process by means of a closure condition. Define two traces as equivalent if they are the same after removal of all hiatons. A free-step process is just one that respects this equivalence. Thus synchronous and asynchronous processes have the same normal form, and differ only in their closure conditions. We will now use the same strategy to define buffered asynchrony, as found in deterministic data-flow, but with an ordering instead of an equivalence

Let c be a channel used for output by a process P. If the channel is buffered, then any output event occurring at a given time may instead be stored in the buffer, and actually occur at some later time, though necessarily before the next output on the same channel. We define an ordering among traces by which moving an output later makes the trace smaller. For example,

$$s\langle c!x, a\rangle t \geq s\langle a, c!x\rangle t, \quad \text{provided } a \text{ is not an output on } c.$$

Variations on this definition are interesting. For example, omission of the proviso would permit the channel c to reorder messages in transit.

A process is said to be output buffered on c if it respects this ordering. Thus

$$P/(s\langle c!x, a\rangle) \text{ is contained in } P/(s\langle a, c!x\rangle)$$

The inclusion cannot be replaced by an equation, because the left hand side will be empty if P cannot output x until after a. For example, a may be an input of a value on which the output value depends.

Input buffering is similarly defined, by moving inputs earlier. A fully asynchronous process is one in which all channels are buffered. But by leaving some channels unbuffered, we have achieved our goal of combining synchronisation and asynchrony in a single normal form, satisfying the relevant closure condition. Unfortunately, we cannot hide the internal channels of a dataflow network. The obvious way to do this is to equate each internal communication event with the hiaton. Unfortunately, I think this again introduces the spectre of non-determinism, which we must now lay.

even

Non-determinism

The description given above has been greatly simplified by considering only the deterministic case. A non-deterministic process is most simply regarded as a set of deterministic processes. One of them will be selected as the actual behaviour of the process, but we do not know and cannot control which of them it will be. So we had better not care. This definition of non-determinism takes advantage of all the mathematics developed for the deterministic case. All operations of the deterministic algebra are defined pointwise on the elements of the set. But sometimes a new operation on a single deterministic process may produce a set of outcomes instead of a singleton. It is this that enables us to give realistic definitions of closure conditions, like that used above to make parallel composition associative.

more

.....to be continued.

most of the familiar process operators of an unbuffered calculus do not respect the simulation