Reasoning About a Distributed Probabilistic System

Ukachukwu Ndukwu[1] **J. W. Sanders**[2] †

[1] Department of Computing,
Division of Information and Communication Sciences
Macquarie University,
NSW, 2109, Sydney, Australia.
Email: ukndukwu@ics.mq.edu.au

[2] International Institute for Software Technology
United Nations University
Macao SAR, China.
Email: jeff@iist.unu.edu

Abstract

Reasoning about a distributed system that exhibits a combination of probabilistic and temporal behaviour does not seem to be easy with current techniques. The reason is the interaction between probability and abstraction (local block), made worse by remote synchronisation. The formalism of process algebra has not so far provided much insight, and so the alternative of shared-variable concurrency has been explored. In this paper the recently proposed language PTSC (for probability, time and shared-variable concurrency) is extended by constructs for interleaving and local block. Both enhance a designer's ability to modularise a design; the latter also permits a design to be compared with its more abstract specification, by concealing appropriately chosen design variables. Laws of the extended language are studied and applied in a case study consisting of a faulty register-transfer-level design.

Keywords: Probabilistic system, shared-variable concurrency, distributed system, event-triggered guarded choice.

1 Introduction

Probabilistic programs afford one means of both efficiency and elegance. In a situation where error can be tolerated (and in many probabilistic programs the error can be decreased at the expense of longer execution time — like further iterations of a loop) then there may exist a probabilistic program that is far more efficient than the best non-probabilistic programs. Primality testing was the original example, but there are now very many more (Motwani & Raghavan 1995). For elegance: in some situations there may be a symmetrical efficient probabilistic program where no non-probabilistic symmetrical program exists. Arbitration is an example. But perhaps most important of all: probabilistic programs provide a way of modelling, reasoning about, and simulating faulty systems.

There is a highly satisfactory theory of sequential probabilistic programs in the Dijkstra style, due largely to McIver and Morgan (McIver & Morgan 2005) but built on work of Clare Jones (Jones 1990) and Kozen (Kozen 1983). However for distributed systems it seems that little satisfactory progress has been made. The interaction between (demonic) nondeterminism and probabilism seems more intricate there because of the scope for demonic behaviour.

Current approaches to probabilistic distributed systems reflect the twin approaches to the theory of general distributed systems theory: namely process algebra and shared-variable concurrency. The former theory has been shaped by CSP (Hoare 1985, Roscoe 2005), CCS (Milner 1989) and ACP (Baeten & Weijland 1990). The latter forms the basis for the memory models of programming languages (for example the Java memory model (Manson et al. 2005)) and for reasoning about hardware (for example Verilog (Verilog, Jifeng & Zhu 2000)).

Whilst process algebra takes for granted the synchronisation mechanism used between processes, in shared-variable concurrency it must be made explicit. For that reason shared-variable concurrency is often quoted as being more difficult to reason about. We demonstrate that difference by including in our case study a description in probabilistic process algebra. However the case study proper, and all reasoning, is performed in our extension to PTSC.

Both process algebra and shared-variable concurrency are well developed but in slightly different directions. Process algebra offers hierarchies of semantic models and complete families of sound laws. However the incorporation of probability there exhibits unsatisfactory features (Morgan et al. 1996); the incorporation of time is intricate (see the survey by Ouaknine and Schneider (Ouaknine & Schneider 2006)); and the two together is problematic. For shared-variable concurrency, on the other hand, much effort has been spent on formalism for reasoning (for a thorough summary see the text by de Roever *et al.* (de Roever et al. 2001)) and less on semantic models. Indeed, since each program has access to the variables of all the other programs, there is scope for mayhem (as unprincipled as unstructured low-level programming). Important methods have been proposed by Owicki and Gries (non-interference (Owiki & Gries 1976)), Cliff Jones (rely and guarantee (Jones 1981)), Misra and Chandy (assumption and commitment (Misra & Chandy 1981)), Brookes (Brookes 1993) and de Roever *et al.* (temporal orderings (de Roever et al. 2001)). This variety of methods attests to the subtlety of the problem.

† The second author acknowledges support from the Macao Science and Technology Development Fund under the PEARL project, grant number 041/2007/A3.

In this paper we consider shared-variable concurrency, and build on the recent model of Zhu, Qin, He and Bowen (Zhu et al. 2006). They propose a language, PTSC, containing probability, time and shared-variable concurrency. Since concurrency is achieved by the interleaving of atomic events, rather than by synchronisation or message passing as in process algebra, it is important to decide in any such model how various threads interact: at which points a thread can be interspersed by another. In (Zhu et al. 2006) that is achieved by a structured operational semantics (Plotkin 2004) with four kinds of transition (corresponding to the progress of an atomic action, time, resolution of nondeterminism, and the triggering of a guarded action). Bisimulation is then used to establish laws. However there the authors did not consider any probability-free operator for combining programs in parallel, of the kind one would expect to use in modularising a system using parallel components. Nor did they consider local block, of use for controlling the scope of variables in interleaving programs and for enabling a low-level program to be compared, after concealing variables appropriately, with a higher-level specification.

A shared-variable-concurrency system is thought of as a single program whose variables are distributed amongst its subprograms; some are unique to a subprogram and play the part of its local variables; some are shared and so are used for communication between subprograms. The language PTSCadded combinators for probability and time, the latter essentially as a global variable. Here, concentrating on probability, we extend PTSCin two ways. Firstly, to enable a system to be modularised in terms of its subprograms executing concurrently, we add an operator for (non-probabilistic) parallel composition. That is essential in order to reason about a system on the basis of the behaviours of its subprograms. Secondly, to allow an implementation to be compared with its more abstract specification, we add local blocks. That is essential in order for an implementation to be verified.

In this paper we add interleaving and local block to PTSC. We choose to present laws without providing an operational semantics (though in fact some of the laws have been confirmed by bisimulation from a structured operational semantics). That is no deficiency because laws also document the manner in which one program can intersperse another. In other words, the computational intuition so often used to justify an operational semantics is equally evident in the laws. And an operational model anyway provides no guarantee of consistency of the laws it posits, in the way that a denotational model does.

We attempt to avoid the difficult problem of reasoning about shared-variable concurrency by using the laws to reason algebraically. A register-transfer-level program is offered as an example, first to test the expressive power of our extension to PTSC and secondly to explore how laws might be used to establish that an implementation with probabilistic behaviour conforms to its probabilistic specification.

In the next section, Section 2, we give an outline of PTSC. We use slightly different syntax from the original (Zhu et al. 2006) in order to conform to what might be called the Dutch style, that is beneficial for calculation; for example the expression

$$(+\ n \in \mathbb{N} : even(n) : x_n)$$

represents the summation $\sum_{n\in 2\mathbb{N}} x_n$. Section 3 introduces interleaving, Section 4 introduces local block and Section 5 presents the case study. In concluding we propose further work.

$\mathbf{skip}$	inaction
$x := e$	assignment
$P \lhd b \rhd Q$	conditional
$\mathbf{do}\ b\ \rightarrow\ P\ \mathbf{od}$	iteration
$P\ ;\ Q$	sequential composition
$P \sqcap Q$	demonic nondeterminism
$(@i \in I : b_i : x_i := e_i\ ;\ P_i)$	guarded assignment
$\#n$	delay
$P\ {}_r\!\oplus\ Q$	binary probabilistic choice
$P\ {}_r\!\parallel\ Q$	probabilistic parallel
$P \mid\mid\mid Q$	interleaving
${}_x[\![\, x{:}X \circ P \,]\!]_x$	local block

Figure 1: The language PTSC augmented by constructs for interleaving and local block (with mild notational variations on the syntax of (Zhu et al. 2006)). We also exploit recursion and mutual recursion when convenient in defining programs.

2 Language PTSC

The language PTSC is based on Dikstra's guarded-command language (Dijkstra 1975), as reflected in the first six constructs of Figure 1. Command **abort** is of course present as **do** *true* $\rightarrow$ **skip od**, and so we follow the authors of (Zhu et al. 2006) and leave it implicit since our concerns here do not address it. On the other hand **skip**, which could equally well have been left implicit as $x := x$, is required here.

A *guarded assignment*, $(@i \in I : b_i : x_i := e_i\ ;\ P_i)$, assumes an index set I, predicates b_i on state space (that need be neither exhaustive nor pairwise disjoint) and a program with a leading assignment of expression e_i to variable x_i and body P_i (which may of course be **skip**). If a guard holds the subsequent program may be scheduled for execution (so nondeterminism arises from overlapping guards); but otherwise the guarded assignment lets time advance. Execution cannot be interspersed between evaluation of the guard and execution of the leading assignment, but of course can be after execution of that assignment. We allow several guarded assignments to be combined with the @ combinator. For example this program waits until either the value of a is 2 or that of b is 1 and then performs the corresponding assignment.

$$(@ : a = 2 : x := 0)\ @\ (@ : b = 1 : y := 3)$$

A *delay*, $\#n$, (for $n : \mathbb{N}$) lets time advance n steps before it terminates. By using sequential composition a *delay-guarded program* is defined

$$\#n\,P\ :=\ \#n\ ;\ P$$

Evidently it can also be constructed from $\#1$ and P.

A *binary probabilistic choice*, $P\ {}_r\!\oplus\ Q$, selects program P with probability r, and program Q with probability $1-r$. The expression r is assumed to be a function of state with values in the real unit interval $[0,1]$. For convenience, we write $1-r$ as $\overline{r}$. The program cannot be interrupted between evaluation of r and execution of the first atomic event of the selected component.

A *probabilistic parallel composition*, $P\ {}_r\!\parallel\ Q$, interleaves (any) initial atomic actions of P and Q with probabilities r and $\overline{r}$ respectively. If the initial atomic events are offered as a probabilistic choice, the resulting probabilities of their being offered for scheduling represent their conditional probabilities.

To deal with multi-way probabilistic choices we adopt the notation $(\oplus i \in I : p_i : P_i)$ in which the probabilities sum to 1: $(+i \in I :: p_i) = 1$. The binary probabilistic choice $P_0 {}_r\!\oplus P_1$ may be expressed in that notation with index set $I := \{0, 1\}$ and probabilities $p_0 := r$, $p_1 := \overline{r}$.

The language PTSC is enriched with a *guarded choice* as follows. Components with leading assignments may be expressed as guarded assignments and combined with (multi-way) probabilistic choice. In the program

$$[]_{i \in I}\{[p_i]\ choice_{j \in J_i}(b_{ij} \& (x_{ij} := e_{ij})\ P_{ij})\}$$

from (Zhu et al. 2006) it is assumed that for each $i : I$ the predicates b_{ij} partition unity (*i.e.* are exhaustive

$$(\forall i : I :: \exists j : J_i \cdot b_{ij})$$

and pairwise disjoint)

$$(\forall i : I \cdot \forall j, j' : J_i : j \neq j' : \neg(b_{ij} \wedge b_{ij'}))$$

and the probabilities sum to 1. Thus in any state, for each $i \in I$, exactly one of the b_{ij} holds (for $j \in J_i$); between the $i \in I$, program $x_{ij} := e_{ij} \fatsemi P_{ij}$ is chosen for (potential) scheduling with probability p_i. In our notation that guarded choice is expressed as a combination of probabilism and guarded assignment

$$(\oplus i \in I : p_i : (@j \in J_i : b_{ij} : x_{ij} := e_{ij} \fatsemi P_{ij}))\,.$$

No action can intersperse between evaluation of the probability, the guard and completion of the leading assignment.

Guarded choices are not combined indiscriminately. For example since the guards in a guarded choice are exhaustive, the passage of time (initially) plays no part in it, and there is no point in combining it with a delay. However if a guarded choice is combined with a guarded assignment (the disjunction of whose guards, it will be recalled, need not be exhaustive), the latter has priority; so only outside the disjunction of the guards of the guarded assignment does the guarded choice apply. Similarly a choice between a delay and a guarded assignment attaches priority to the guarded assignment.

In the presence of probabilistic behaviour, programs are regarded as the same if their behaviours are the same to within unit probability (rather than 'absolutely'). For example a random walk on the integers with equal probability of incrementing and decrementing its value n has probability 1 of returning to the origin, although there is an execution that increments forever. So a loop with guard $n \neq 0$ and body that increments and decrements n with equal probability, is for us the same as the assignment $n := 0$.

As a result, a conditional is a special case of a probabilistic choice. Indeed for predicate b on state space,

$$P \lhd b \rhd Q \quad = \quad P\ {}_{[b]}\!\oplus Q$$

where $[b]$ is the function on state space with value 1 if b and 0 if $\neg b$. So laws for conditional may be inferred from those for probabilism.

In a shared-variable program, each component program can in principle read and write-to all variables. However in practice that is seldom necessary. Practical designs exploit severe modularisation in which component programs access variables selectively. We write $vars(P)$ for the set of variables that program P accesses (either reads or writes-to).

For example here is a PTSC program C (similar to one that will arise in Section 5). $vars(C) = \{a, b, c, d, e\}$ where all variables are of type $\mathbb{N}$ except e which is Boolean. After updating variable a, the program enters a loop in which it waits for variables b and c to be 0. When they are, it reads the value of variable Boolean e; if high, it updates b and c with equal likelihood and iterates; if low, it updates d and returns to the start of C.

$$\begin{array}{lll} C & := & a := 2 \fatsemi \\ & & \textbf{do}\ true\ \rightarrow \\ & & \quad (@ : (b = 0 \wedge c = 0) : \\ & & \quad ((b := 2\ {}_{\frac{1}{2}}\!\oplus c := 1) \\ & & \quad \lhd e \rhd \\ & & \quad (d := 1 \fatsemi C))) \\ & & \textbf{od} \end{array}$$

The last two lines of Figure 1 introduce the constructs added here to PTSC. They are introduced as follows.

3 Interleaving

Programs that communicate *via* shared variables execute in parallel, interleaving their atomic actions. The laws for interleaving, $|||$, make precise the points in a program at which another may intersperse its own atomic actions. That idea goes back, as with so much, to a paper of Dikjstra, in this case (Dijkstra 1965) in which he exploited concurrency with **parbegin**, **parend** blocks.

If $x := e$ and $y := f$ are atomic assignments to distinct variables then their concurrent execution must achieve one before the other, yet the order is not determined

$$\begin{array}{l} x := e \,|||\, y := f \\ = \\ (x := e \fatsemi y := f) \sqcap (y := f \fatsemi x := e) \end{array} \tag{1}$$

Evidently the result depends, in general, on the order (since an updated value of one variable may be used in the other expression). This situation is thus usually to be avoided (even if $x = y$). By comparison the *multiple assignment*,

$$x, y := e, f\,,$$

evaluates the two assignments atomically, concurrently, and without interference between the expressions (by taking a copy of each of x and y, for use in evaluating the expressions, just before the assignments begin it can be ensured that the initial and final values are related thus: $x = e[x_0, y_0/x, y] \wedge y = f[x_0, y_0/x, y]$).

The appropriate law for interleaving is

$$\begin{array}{l} (x := e \fatsemi P) \,|||\, (y := f \fatsemi Q) \\ = \\ x := e \fatsemi (P \,|||\, (y := f \fatsemi Q)) \\ \sqcap \\ y := f \fatsemi ((x := e \fatsemi P) \,|||\, Q)\,. \end{array} \tag{2}$$

The interleaving of a program P with inaction consists just of the actions of P.

$$\textbf{skip} \,|||\, P \quad = \quad P \tag{3}$$

Interleaving attributes no importance to the order of its component programs (it is commutative) nor to the order in which more than two are combined (it is associative).

$$P \,|||\, Q \quad = \quad Q \,|||\, P \tag{4}$$

$$P \,|||\, (Q \,|||\, R) \quad = \quad (P \,|||\, Q) \,|||\, R \tag{5}$$

The interleaving of a program P with a nondeterministic choice between programs Q and R consists of the interleaving of P with either Q or R, the choice being again nondeterministic.

$$P \,|||\, (Q \sqcap R) \;=\; (P \,|||\, Q) \sqcap (P \,|||\, R) \qquad (6)$$

Similar reasoning leads to laws for probabilistic choice and probabilistic parallel, provided that execution of R does not change the probabilistic expression r which is evaluated first on the right-hand side.

$$(P \,{}_r\!\oplus\, Q) \,|||\, R \;=\; (P \,|||\, R) \,{}_r\!\oplus\, (Q \,|||\, R) \qquad (7)$$
$$(P \,{}_r\!\parallel\, Q) \,|||\, R \;=\; (P \,|||\, R) \,{}_r\!\parallel\, (Q \,|||\, R) \qquad (8)$$

The laws for guarded assignment ensure that no action can be interspersed between the test of a guard which is true and the assignment it guards.

$$\begin{array}{l}(@i \in I : b_i : x_i := e_i \fatsemi P_i) \\ |||\\ (@j \in J : c_j : y_j := f_j \fatsemi Q_j) \\ = \\ (@i \in I, j \in J : b_i \wedge c_j : x_i := e_i \fatsemi \\ (P_i \,|||\, (y_j := f_j \fatsemi Q_j)) \\ \sqcap \\ y_j := f_j \fatsemi ((x_i := e_i \fatsemi P_i) \,|||\, Q_j)) \\ @ \\ (@i \in I, j \in J : b_i \wedge \neg c_j : x_i := e_i \fatsemi \\ (P_i \,|||\, (y_j := f_j \fatsemi Q_j))) \\ @ \\ (@i \in I, j \in J : \neg b_i \wedge c_j : y_j := f_j \fatsemi \\ ((x_i := e_i \fatsemi P_i) \,|||\, Q_j)))\end{array} \qquad (9)$$

Priority is given to a guarded assignment: an unguarded assignment in parallel has the chance to be scheduled only if no guard holds.

$$\begin{array}{l}(@i \in I : b_i : x_i := e_i \fatsemi P_i) \,|||\, (y := f \fatsemi Q) \\ = \\ ((@i \in I : b_i : x_i := e_i \fatsemi (P_i \,|||\, (y := f \fatsemi Q)) \\ \lhd\, \exists i : I \cdot b_i \,\rhd \\ y := f \fatsemi (@i \in I : b_i : (x_i := e_i \fatsemi P_i) \,|||\, Q))\end{array} \qquad (10)$$

The passage of time is not interleaved, but experienced equally by both component programs.

$$(\#1\, P) \,|||\, (\#1\, Q) \;=\; \#1\, (P \,|||\, Q) \qquad (11)$$

The interleaving of a delay with a guarded assignment gives priority to the guarded assignment and only outside the disjunction of its guards is time allowed to advance.

$$(\#1\, P) \,|||\, (y := f \fatsemi Q) = (y := f) \fatsemi (\#1\, P \,|||\, Q) \qquad (12)$$

$$\begin{array}{l}(\#1\, P) \,|||\, (@i \in I : b_i : x_i := e_i \fatsemi P_i) \\ = \\ ((@i \in I : b_i : x_i := e_i \fatsemi (\#1\, P \,|||\, P_i)) \\ \lhd\, \exists i : I \cdot b_i \,\rhd \\ x_i := e_i)\end{array} \qquad (13)$$

4 Local Block

Local blocks are used to delimit the scope of variables. Our notation for a block in which the variable x, of type X, is local in program Q is

$$_x[\![\, x : X \circ Q \,]\!]_x \,.$$

We treat a local block as a whole, matching the beginning of the block with its end. An exposition in the setting of *wp* and the refinement calculus is Morgan's textbook (Morgan 1994). For an alternative treatment, separating the two halves of a block, see Hoare and He's UTP text (Hoare & Jifeng 1998) (noting that the separation is independent of their UTP treatment of it).

Many of the laws are standard, even if they are not well advertised. For example, a localised variable is invisible outside its block.

$$_x[\![\, x : X \circ x := e \,]\!]_x \;=\; \mathbf{skip} \qquad (14)$$

As a result, if x does not appear in program P then since it plays no part in P, even if it is initialised, its localisation leaves P unchanged.

$$_x[\![\, x : X \circ P \,]\!]_x \;=\; P \qquad (15)$$
$$_x[\![\, x : X \circ x := e \fatsemi P \,]\!]_x \;=\; P \qquad (16)$$

Localising a nondeterministic choice between two programs gives a localised version of one or the other, the choice being again nondeterministic.

$$\begin{array}{l}{}_x[\![\, x : X \circ P \sqcap Q \,]\!]_x \\ = \\ {}_x[\![\, x : X \circ P \,]\!]_x \;\sqcap\; {}_x[\![\, x : X \circ Q \,]\!]_x\end{array} \qquad (17)$$

We assume that law also for arbitrary (nonvoid) nondeterminism.

Our treatment assumes that a block begins with a type declaration for the new local variable, but not necessarily with its initialisation. An uninitialised local variable assumes a nondeterministically-chosen value of its type. Thus,

$$x : X \circ P \;=\; x : X \circ \sqcap_{v \in X}\; x := v \fatsemi P \,. \qquad (18)$$

That involves finite nondeterminism iff the type X is finite. Of course typically the type declaration is followed immediately by initialisation in P, which 'supersedes' that nondeterministic choice. For example if x is a Boolean variable and P has a leading assignment initialising x to *false*, then the nondeterministic default is indeed superseded:

$$(\sqcap_{v:\mathbb{B}}\; x := v \fatsemi x := \mathit{false}) \;=\; (x := \mathit{false}) \,.$$

The companion law to (16) can now be stated. Assume $y \neq x$. If the local variable x is not free in expression e and y does not appear in program P then

$$\begin{array}{l}{}_x[\![\, x : X \circ y := e \fatsemi P \,]\!]_x \\ = \\ y := e \fatsemi {}_x[\![\, x : X \circ P \,]\!]_x \,.\end{array} \qquad (19)$$

To deal with the case x free in e, if x is initialised (and $x \neq y$ and y does not appear in program P) then

$$\begin{array}{l}{}_x[\![\, x : X \circ x := x_0 \fatsemi y := e \fatsemi P \,]\!]_x \\ = \\ y := e[x_0/x] \fatsemi {}_x[\![\, x : X \circ x := x_0 \fatsemi P \,]\!]_x \,.\end{array} \qquad (20)$$

If local variable x is not initialised then Law (18) applies. Assuming the extension of Law (17) to arbitrary nondeterminism (and $x \neq y$ and y not in program P) yields

$$\begin{array}{l}{}_x[\![\, x : X \circ y := e \fatsemi P \,]\!]_x \\ = \\ \sqcap_{v \in X}\; y := e[v/x] \fatsemi {}_x[\![\, x : X \circ P \,]\!]_x \,.\end{array} \qquad (21)$$

Because a localised variable is invisible outside its block, localising it immediately a second time has no further effect.

$$_x[\![\, x : X \circ {}_x[\![\, x : X \circ P \,]\!]_x \,]\!]_x \;=\; {}_x[\![\, x : X \circ P \,]\!]_x \qquad (22)$$

The localisation of (distinct) variables can be achieved in either order.

$$ {}_x[\![\, x{:}X \circ {}_y[\![\, y{:}Y \circ P \,]\!]_y \,]\!]_x \quad = \quad {}_y[\![\, y{:}Y \circ {}_x[\![\, x{:}X \circ P \,]\!]_x \,]\!]_y \tag{23} $$

That means the definition of block can be extended from one local variable to any finite set of local variables

$$ {}_{\{\}}[\![\, P \,]\!]_{\{\}} \;:=\; P $$

$$ {}_{x,y}[\![\, x{:}X \fatsemi y{:}Y \circ P \,]\!]_{x,y} \;:=\; {}_x[\![\, x{:}X \circ {}_y[\![\, y{:}Y \circ P \,]\!]_y \,]\!]_x \,. $$

Indeed Laws (22) and (23) ensure the consistency of that definition: blocks do indeed behave as a 'set-like' function of their local variables. They demonstrate, for example, that the block with local variables $\{x, x, y\}$ is the same, just as it ought to be, as that with local variables $\{y, x\}$.

Recall (from Section 2) that since a conditional is a special case of a probabilistic choice, the laws for localising a conditional are subsumed in those for localising probabilism.

If x is not free in the probabilistic expression r then localising a probabilistic choice between P and Q results in a probabilistic choice between the localised versions of P and Q.

$$ {}_x[\![\, x{:}X \circ P \,{}_r\!\oplus\, Q \,]\!]_x \quad = \quad {}_x[\![\, x{:}X \circ P \,]\!]_x \;{}_r\!\oplus\; {}_x[\![\, x{:}X \circ Q \,]\!]_x \tag{24} $$

If the probability r depends on an initialised local variable x then the probabilism is resolved by substitution (as in Law (20)).

$$ {}_x[\![\, x{:}X \circ x := x_0 \fatsemi P \,{}_r\!\oplus\, Q \,]\!]_x \quad = \quad {}_x[\![\, x{:}X \circ x := x_0 \fatsemi P \,]\!]_x \;{}_{r[x_0/x]}\!\oplus\; {}_x[\![\, x{:}X \circ x := x_0 \fatsemi Q \,]\!]_x \tag{25} $$

But if the local variable is uninitialised then the default is a nondeterministic choice over all possible initialisations (Law (21))

$$ {}_x[\![\, x{:}X \circ P \,{}_r\!\oplus\, Q \,]\!]_x \quad = \quad \sqcap_{v \in X} \; {}_x[\![\, x{:}X \circ P \,]\!]_x \;{}_{r[v/x]}\!\oplus\; {}_x[\![\, x{:}X \circ Q \,]\!]_x \,. \tag{26} $$

For example we can infer that a conditional with an unitialised Boolean test is nondeterministic:

$$ \begin{array}{ll} & {}_x[\![\, x{:}\mathbb{B} \circ P \lhd x \rhd Q \,]\!]_x \\ = & \qquad \text{Law (18)} \\ & {}_x[\![\, x{:}\mathbb{B} \circ (\sqcap_{v \in \mathbb{B}} \; x := v) \fatsemi (P \lhd x \rhd Q) \,]\!]_x \\ = & \qquad \text{standard} \\ & {}_x[\![\, x{:}\mathbb{B} \circ \sqcap_{v \in \mathbb{B}} \; (x := v \fatsemi (P \lhd x \rhd Q)) \,]\!]_x \\ = & \qquad \text{standard} \\ & {}_x[\![\, x{:}\mathbb{B} \circ \sqcap_{v \in \mathbb{B}} \; (P \lhd x[v/x] \rhd Q) \,]\!]_x \\ = & \qquad \text{calculus} \\ & {}_x[\![\, x{:}\mathbb{B} \circ (P \lhd \mathit{true} \rhd Q) \sqcap (P \lhd \mathit{false} \rhd Q) \,]\!]_x \\ = & \qquad \text{definition of conditional} \\ & {}_x[\![\, x{:}\mathbb{B} \circ P \sqcap Q \,]\!]_x \\ = & \qquad \text{Law (17)} \\ & {}_x[\![\, x{:}\mathbb{B} \circ P \,]\!]_x \;\sqcap\; {}_x[\![\, x{:}\mathbb{B} \circ Q \,]\!]_x \,. \end{array} $$

Probabilistic parallel composition is similar to probabilistic choice. Its analogue of (24), for example, is (provided x is not free in r),

$$ {}_x[\![\, x{:}X \circ P \,{}_r\!\parallel\, Q \,]\!]_x \quad = \quad {}_x[\![\, x{:}X \circ P \,]\!]_x \;{}_r\!\parallel\; {}_x[\![\, x{:}X \circ Q \,]\!]_x \,. \tag{27} $$

Again, guarded choice follows a similar pattern. The simple case, in which the guards do not depend on the local variable, is

$$ {}_x[\![\, x{:}X \circ (@i \in I : b_i : P_i) \,]\!]_x \quad = \quad (@i \in I : b_i : {}_x[\![\, x{:}X \circ P_i \,]\!]_x) \,. \tag{28} $$

Time is not localisable.

$$ {}_x[\![\, x{:}X \circ \#n \, P \,]\!]_x \;=\; \#n \; {}_x[\![\, x{:}X \circ P \,]\!]_x \tag{29} $$

The localisation of an interleaving of P and Q interleaves the localisations of P and Q.

$$ {}_x[\![\, x{:}X \circ P \mid\mid\mid Q \,]\!]_x \quad = \quad {}_x[\![\, x{:}X \circ P \,]\!]_x \mid\mid\mid {}_x[\![\, x{:}X \circ Q \,]\!]_x \tag{30} $$

5 Case Study

In this section we consider a circuit, at the register-transfer level, that adds its two integer inputs *inx* and *iny* and outputs their sum to variable *out*. However the register containing *inx* is faulty. Our first task is to model the circuit accurately in PTSC and our second is to reason about its behaviour.

5.1 Informal Description

We assume that the environment of the circuit supplies the two inputs when the circuit is ready for them, and receives the single output when the circuit is ready to provide it. The circuit is abstract in the sense that the registers that hold the two inputs are of type natural number (rather than being bit strings, and rather than assuming the numbers to be *a priori* bounded). One register, X performs natural-number increment and the other, Y the decrement of a positive natural number. The circuit attempts to achieve the required addition by successive concurrent faulty-increments and decrements until the decrement register contains zero, at which point its initial value ought (in the absence of faults) to have been 'transferred' to the other register.

The specification thus has three variables, all integers. When the circuit behaves correctly, it achieves the sum:

$$ \mathit{Spec} \;:=\; \mathit{inx}, \mathit{iny}, \mathit{out} : \mathbb{N} \circ \; \mathit{out} := \mathit{inx} + \mathit{iny} \,. $$

The implementation contains five local variables plus the two register variables. It consists of an initialisation of the local variables, followed by the interleaving of the increment register X, the decrement register Y and the control unit C.

$$ \begin{array}{rl} \mathit{Imp} \;:=\; & x, y : \mathbb{N} \fatsemi \\ & \mathit{ie}, \mathit{op} : 0 \mid 1 \mid 2 \fatsemi \\ & \mathit{oe}, f : 0 \mid 1 \fatsemi \\ & s : \mathbb{B} \circ \\ & \mathit{ie}, \mathit{op}, \mathit{oe}, f := 0, 0, 0, 1 \fatsemi \\ & X \mid\mid\mid Y \mid\mid\mid C \end{array} $$

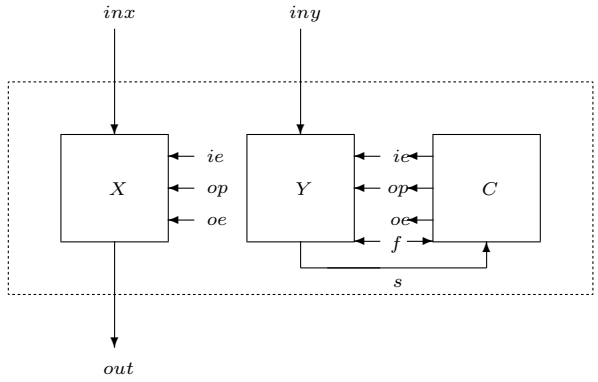

Figure 2: Register-transfer circuit interpreted as a data-flow design. The enables ie, op and oe are shared variables essentially written-to by C and read by X and Y; but to avoid races they also act as semaphores and so are written also by the two registers. The status signal s, however, is written-to by Y and read by C. The semaphore f is written-to and read by both Y and C.

The three component programs share variables as follows. Recalling (Section 2) the notation for the variables accessed by a program:

$$\begin{array}{lcl} vars(X) & := & \{inx, x, ie, op, oe, out\} \\ vars(Y) & := & \{iny, y, ie, op, oe, s, f\} \\ vars(C) & := & \{ie, op, oe, s, f\}\,. \end{array}$$

Thus ie, op and oe are common to all three while f and s to just Y and C. Variables inx and out are common to X and the environment and variable iny is common to Y and the environment. However x and y are unique to their programs. See Figure 2.

However the increment register X increments x with probability r but overlooks the increment with probability $\overline{r}$.

We must first express X, Y and C in PTSC. Then we must establish the behaviour of ${}_E[\![\, Imp \,]\!]_E$, where $E := \{x, y, ie, op, oe, s, f\}$, and relate it to $Spec$.

5.2 Definition in PTSC

The increment register X consists of a ternary guarded choice, determined by the (pairwise) disjoint conditions $ie = 2$, $op = 2$ and $oe = 1$. If the first guard holds, it inputs inx, decrements ie (to avoid repeated inputs) and iterates. If the second guard holds, it tries to increment its x register but does so successfully with probability r whilst with probability $\overline{r}$ it skips; then it decrements op (again, to avoid unwanted—this time critical—repetition) and iterates. We assume that r is a positive constant, so the faulty register always has the same positive chance of behaving correctly. If the third guard holds it outputs the value of x and terminates.

$$\begin{array}{lcl} X & := & (@ : ie = 2 : x := inx \fatsemi ie := ie - 1 \fatsemi X)\ @ \\ & & (@ : op = 2 : (x := x + 1\ {}_r\!\oplus \mathbf{skip}) \fatsemi \\ & & op := op - 2 \fatsemi X)\ @ (@ : oe = 1 : out := x) \end{array}$$

The decrement register Y consists of the same guarded choice. In the case $ie = 2$ it begins like X by inputting to y and decrementing ie; but then it writes the Boolean $(y \neq 0)$ to s, lowers the flag f and iterates. In the case $op = 2$ it decrements its y register and op then awaits the semaphore f; when $f = 1$ it updates s to the usual Boolean, lowers f and iterates. In the third case, $oe = 1$, it terminates.

$$\begin{array}{lcl} Y & := & (@ : ie = 2 : y := iny \fatsemi ie := ie - 1 \fatsemi \\ & & s := (y \neq 0) \fatsemi f := 0 \fatsemi Y)\ @ \\ & & (@ : op = 2 : y := y - 1 \fatsemi op := op - 2 \fatsemi \\ & & (@ : f = 1 : s := (y \neq 0) \fatsemi f := 0) \fatsemi Y)\ @ \\ & & (@ : oe = 1 : \mathbf{skip}) \end{array}$$

The control unit C begins by enabling register inputs by setting $ie = 2$. It then enters a loop in which it waits for the semaphore f to be high and any previous op action to be complete ($op = 0$) before reading the Boolean s and, if $y \neq 0$ enabling op and setting the semaphore $f = 1$; otherwise it enables output and terminates.[1] It seems slightly more natural (and facilitates comparison in the next section) to use (tail) recursion rather than a loop with guard $true$, as follows.

$$\begin{array}{lcl} C & := & ie := 2 \fatsemi D \\ D & := & (@ : (f = 0 \wedge op = 0) : \\ & & (op := 2 \fatsemi f := 1 \fatsemi D) \lhd s \rhd oe := 1) \end{array}$$

We conclude that whilst the design can be expressed in PTSC, the use of various values (ie, op, oe and f) as clocking variables, or semaphores, betrays the existence of a simpler more abstract representation. For the purposes of informing the PTSC description, we consider it now.

5.3 Comparison with Process Algebra

In process algebra, channels can be used to communicate the status value s from Y to the control unit C and the values of the enables ie, op and oe from C to the registers. However the occurrence of probability requires us to use probabilistic process algebra. We choose the process algebra CSP (Roscoe 2005) and its probabilistic extension pCSP (Morgan et al. 1996), which contains the binary combinator ${}_r\!\oplus$ for probabilistic choice between its two arguments.

We recall that in CSP an assignment is a process, and so is combined with its successor process using $\fatsemi$, whilst an event is combined with its successor process using $\rightarrow$. In CSP we continue to write the conditional construct 'A if b else C' as $A \lhd b \rhd C$.

The increment register X offers its environment a choice of three enables and in each case performs the expected actions

$$\begin{array}{lcl} X & := & ie \rightarrow x := inx \fatsemi X \\ & & [\!] \\ & & op \rightarrow (x := x + 1\ {}_r\!\oplus \mathbf{skip}) \fatsemi X \\ & & [\!] \\ & & oe \rightarrow out := x\,. \end{array}$$

The decrement register Y offers its environment a choice of the three enables, but also communication along channel s on which it outputs the Boolean t

$$\begin{array}{lcl} Y & := & ie \rightarrow y := iny \fatsemi Y \\ & & [\!] \\ & & op \rightarrow y := y - 1 \fatsemi Y \\ & & [\!] \\ & & s!t \rightarrow Y \\ & & [\!] \\ & & oe \rightarrow \mathbf{skip}\,. \end{array}$$

The control unit C starts by enabling ie and then iteratively inputs the value from channel s and, if it

[1] The simple modification which returns to the start, ready for another cycle, requires '$\fatsemi C$' after the assignment to oe in C and similarly in X and Y. However it adds unnecessarily to the example.

is true enables op and reiterates but, if it is false then enables oe and terminates

$$C \;:=\; ie \;\rightarrow\; \mu D \cdot (s?t \;\rightarrow\; ((op \rightarrow D) \lhd t \rhd (oe \rightarrow \mathbf{skip}))) \,.$$

The implementation is as before

$$Imp \;:=\; (X \| Y \| C)_{vars(X) \cap vars(Y) \cap vars(C)} \setminus E$$

where we have been slightly cavalier with CSP notation in combining the hiding of the events with those of the internal variables.

The synchronisation between sender and receiver that is part of CSP obviates the need for the semaphores that cluttered our PTSC description. In particular Y and C appear simpler, as does the design as a whole.

5.4 Behaviour Identified

In this section we outline an algebraic approach to identifying the behaviour of the design *Imp*. Consider first just the interleaving of C and Y.

$$\begin{array}{l}
ie, op, oe, f := 0, 0, 0, 1 \fatsemi \\
C \,|||\, Y \\
= \qquad\qquad \text{definitions of } C \text{ and } Y \\
ie, op, oe, f := 0, 0, 0, 1 \fatsemi \\
(ie := 2 \fatsemi \mu D \cdot (@ : (f = 0 \wedge op = 0) : \\
(op := 2 \fatsemi f := 1 \fatsemi D) \lhd s \rhd (oe \fatsemi C)) \\
||| \\
(@ : ie = 2 : y := iny \fatsemi ie := ie - 1 \fatsemi \\
s := (y \neq 0) \fatsemi f := 0 \fatsemi Y) \; @ \\
(@ : op = 2 : y := y - 1 \fatsemi op := op - 2 \fatsemi \\
(@ : f = 1 : s := (y \neq 0) \fatsemi f := 0) \fatsemi Y)) \\
= \qquad \text{standard reasoning and Laws (9) and (10)} \\
ie, op, oe, f := 0, 0, 0, 1 \fatsemi \\
ie := 2 \fatsemi y := iny \fatsemi ie := ie - 1 \fatsemi s := (y \neq 0) \fatsemi \\
f := 0 \fatsemi \\
\mu Z \cdot (op := 2 \fatsemi (f := 1 \,|||\, (y := y - 1 \fatsemi \\
\quad op := op - 2)) \fatsemi s := (y \neq 0) \fatsemi f := 0 \fatsemi Z \\
\quad \lhd s \rhd \\
\quad oe := 1)
\end{array}$$

Next, including also X, we find

$$\begin{array}{l}
ie, op, oe, f := 0, 0, 0, 1 \fatsemi \\
(C \,|||\, Y \,|||\, X) \\
= \qquad\qquad \text{definition of } X \text{ and Law (5)} \\
ie, op, oe, f := 0, 0, 0, 1 \fatsemi \\
(ie := 2 \fatsemi y := iny \fatsemi ie := ie - 1 \fatsemi s := (y \neq 0) \fatsemi \\
f := 0 \fatsemi \\
\mu Z \cdot (op := 2 \fatsemi (f := 1 \,|||\, (y := y - 1 \fatsemi \\
\quad op := op - 2)) \fatsemi s := (y \neq 0) \fatsemi f := 0 \fatsemi Z \\
\quad \lhd s \rhd \\
\quad oe := 1) \\
\;|||\; \\
(@ : ie = 2 : x := inx \fatsemi ie := ie - 1 \fatsemi X) \; @ \\
(@ : op = 2 : (x := x +_1 1 \oplus \mathbf{skip}) \fatsemi \\
\;op := op - 2 \fatsemi X) \; @ \;\; (@ : oe = 1 : out := x)) \\
= \qquad \text{standard reasoning and Laws (9) and (10)} \\
op, oe, f := 0, 0, 1 \fatsemi \\
ie := 2 \fatsemi ((x := inx \fatsemi ie := ie - 1) \\
||| \\
(y := iny \fatsemi ie := ie - 1 \fatsemi s := (y \neq 0) \fatsemi f := 0)) \fatsemi \\
\mu Z \cdot (op := 2 \fatsemi (((x := x +_1 1 \oplus \mathbf{skip}) \fatsemi \\
\quad op := op - 2 \fatsemi X) \,|||\, f := 1 \,|||\, (y := y - 1 \fatsemi \\
\quad op := op - 2)) \fatsemi s := (y \neq 0) \fatsemi f := 0 \fatsemi Z \\
\quad \lhd s \rhd \\
\quad oe := 1 \fatsemi out := x) \,.
\end{array}$$

Thus localising the variables in the set $F \;:=\; \{ie, op, oe, f\}$,

$$\begin{array}{l}
{}_F[\![ie, op, oe, f := 0, 0, 0, 1 \fatsemi (C \,|||\, Y \,|||\, X)]\!]_F \\
= \text{ standard reasoning and Laws (14), (19), (20) and (24)} \\
(x := inx \,|||\, (y := iny \fatsemi s := (y \neq 0))) \fatsemi \\
\mu Z \cdot ((((x := x +_1 1 \oplus \mathbf{skip}) \fatsemi X) \,|||\, y := y - 1) \fatsemi \\
\quad s := (y \neq 0) \fatsemi Z \\
\quad \lhd s \rhd \\
\quad out := x) \,.
\end{array}$$

To establish termination (of the tail recursion on Z that is a mildly disguised loop) it suffices to take variant function y.

On each iteration the choice between incrementing and leaving x unchanged is binary with fixed probability r. So after i iterations the value of x is binomially distributed in the interval $[inx, inx + i]$. But the number of iterations is $i = iny - y$ and so we take as loop invariant $0 \leq y \leq iny$ and

$$Inv \;:=\; \forall j : [0, iny - y] \cdot x = inx + j \quad \text{with probability } \tbinom{iny - y}{j} r^j \, \overline{r}^{iny - y - j} \,.$$

Indeed Inv is established: initially $y = iny$ and so $x = inx$ with probability $\binom{0}{0} r^0 \, \overline{r}^{0-0} \;=\; 1$. It is also seen to be maintained, by the expected but slightly longer calculation.

Thus, localising also s

$$\begin{array}{l}
{}_{F \cup \{s\}}[\![ie, op, oe, f := 0, 0, 0, 1 \fatsemi (C \,|||\, Y \,|||\, X)]\!]_{F \cup \{s\}} \\
\quad = \qquad \text{standard reasoning and Laws (14) and (20)} \\
(x, y := inx, iny) \fatsemi out := x \,,
\end{array}$$

where the distribution of values for $out = x$ satisfies the invariant $Inv[out/x]$ and the negation of the guard $(y \neq 0)$:

$$\forall j : [0, iny] \cdot out = inx + j \quad \text{with probability } \tbinom{iny}{j} r^j \, \overline{r}^{iny - j} \,.$$

In other words, as expected, *out* is binomially distributed over the interval $[inx, inx + iny]$ of integers, with 'weight' r.

Finally, localising also x and y gives (with E defined at the end of Section 5.1)

$$\begin{array}{l}
{}_E[\![ie, op, oe, f := 0, 0, 0, 1 \fatsemi (C \,|||\, Y \,|||\, X)]\!]_E \\
= \\
out := inx + iny \,,
\end{array}$$

with binomial distribution of values for *out*, as above.

6 Conclusions

The language PTSC of Zhu, Qin, He and Bowen (Zhu et al. 2006), for probability, time and shared-variable concurrency, has been extended in this paper to incorporate interleaving and local block. The resulting language permits programs to be combined non-probabilistically via interleaving, variables to be localised (essential if the complexities of shared-variable programs have any hope of being subdued) and an implementation to be compared, after localising appropriate low-level variables, with its more abstract specification. Laws have been given for the new operators and a register-transfer-level design has been shown to meet its probabilistic specification. In general, subtle methods are required to reason about shared-variable concurrency (de Roever et al. 2001). Use of law-based reasoning, as done in this case study, seems to offer an alternative approach that may even be able to be automated (Höfner & Struth 2008).

The next step confronting this work is, as for the original PTSC, construction of a denotational model. We should like to have a normal form, and sufficiently many laws to transform any program into it.

Experience with CSP has played an important part in the intuitions behind PTSC, particularly for the guarded constructs. The proximity of the PTSC and CSP realisations of our case study confirms that, and suggests that it may be interesting to translate from (simple) designs in CSP that use channel communications to avoid races, to PTSC designs which require semaphores or equivalents for the same purpose. With more experience we will presumably gain some appreciation of the difference between algebraic reasoning and the various techniques for reasoning about shared-variable interactions. A Galois connection between the models would afford a systematic start. It would be interesting to compare PTSC with pGCL, the probabilistic version of Dijkstra's guarded-command language (McIver & Morgan 2005) under both the expectation-transformer semantics and distributional semantics.

The case study contains just concurrency, locality and probability, overlooking time. However it could easily be extended to contain time: the registers could take different times (modelled in PTSC using delay) to perform their updates and in particular the faulty increment register could take different times depending on whether or not a fault occurs. Perhaps register delays could even be configured to avoid races and hence obviate the need for semaphores (as in hardware optimisation). That constitutes further work. The treatment of time in PTSC is a little simplistic: it behaves just like a global variable. It would be interesting to explore alternatives.

7 Acknowledgement

The referees are acknowledged for helpful comments.

References

`http://www.verilog.com`

Baeten, J. C. M. & Weijland, W. P. (1990), *Process Algebra*, Cambridge Tracts in Theoretical Computer Science, **18**. Cambridge University Press.

Brookes, S. D. (1993), A fully abstract semantics of a shared-variable parallel language. In *Proc. 8th Annual IEEE Symposium on Logic in Computer Science*, 98–109. IEEE Computer Society Press.

Dijkstra, E. W. (1965), Solution of a problem in concurrent programming control. *Communications of the ACM*, **8**(9):569.

Dijkstra, E. W. (1975), Guarded commands, nondeterminacy and the formal derivation of programs. *Communications of the ACM*, **18**:453–457.

He Jifeng & Zhu, H. (2000), Formalising Verilog. In *Proc. ICECS 2000: 3rd IEEE International Conference on Electronics, Circuits and Systems*, 412–415. IEEE Computer Society Press.

Hoare, C. A. R. (1985), *Communicating Sequential Processes*, Prentice-Hall International.

Hoare, C. A. R. & He Jifeng (1998), *Unifying Theories of Programming*, Prentice-Hall Series in Computer Science.

Höfner, P. & Struth, G. (2008), Can refinement be automated? *Electronic Notes in Theoretical Computer Science (ENTCS)*, **201**:197–222.

Jones, C. (1990), Probabilistic Nondeterminism. Ph.D., LFCS, The University of Edinburgh. Monograph ECS-LFCS-90-105.

Jones, C. B. (1981), Development Methods for Computer Programs Including a Notion of Interference. Ph.D., OUCL, The University of Oxford.

Kozen, D. (1983), Semantics of probabilistic programs. *Journal of Computer and System Sciences*, **27**:333–354.

McIver, A. K. & Morgan, C. C. (2005), *Abstraction, Refinement and Proof for Probabilistic Systems*, Monographs in Computer Science, Springer Verlag.

Manson, J. , Pugh, W. & Adve, S. (2005), The Java memory model. In *Principles of Programming Languages (POPL)*, 378–391.

Milner, R. (1989), *Communication and Concurrency*, Prentice-Hall International.

Misra, J. & Chandy, K. M. (1981), Proof of networks of processes. *IEEE Transactions on Software Engineering*, **7**(7):417–426.

Morgan, C. C. (1994), *Programming from Specifications.*, Prentice-Hall International, second edition.

Morgan, C. C. , McIver, A. K. , Seidel, K. & Sanders, J. W. (1996), Refinement-oriented probability for CSP. *Formal Aspects of Computing*, **8**(6):617–647, 1996.

Morgan, C. C. (1996), Proof rules for probabilistic loops. In He Jifeng, Cooke, J. & Wallis, P. editors, *Proceedings of the BCS-FACS 7th Refinement Workshop*. Workshops in Computing, Springer Verlag.

Motwani, R. & Raghavan, R. (1995), *Randomized Algorithms*,Cambridge University Press. Prentice-Hall International, second edition.

Ouaknine, J. & Schneider, S. A. (2006), Timed CSP: a retrospective. Proceedings of the workshop *Essays on Algebraic Process Calculi, (APC 25). Electronic Notes in Theoretical Computer Science*, **162**:273–276.

Owicki, S. & Gries, D. (1976), An axiomatic proof technique for parallel programs. *Acta Informatica*, **6**:319–340.

Plotkin, G. D. (2004), A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming*, **60,61**:17–139, 2004. Originally: Technical Report 19, University of Aarhus Denmark.

de Roever, W.-P. , de Boer, F. , Hannemann, U. , Hooman, J. , Lakhnech, Y. , Poel, M. & Zwiers, J. (2001), *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, Cambridge Tracts in Theoretical Computer Science **54**, Cambridge University Press.

Roscoe, A. W. (2005), *The Theory and Practice of Concurrency*, Prentice-Hall International.

Zhu, H. , Qin, S. , He Jifeng & Bowen, J. P. (2006), Integrating probability with time and shared-variable concurrency. *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, IEEE Computer Society Press, 179–189.