Adding “Process Algebra” to TLA

Leslie Lamport

Sun 22 Jan 1995 [12:44]

This is a rough first draft of some very sweet syntactic sugar for defining TLA actions and associated predicates, inspired by process algebra. The first part describes the notation; the second part contains some handwaving about how one might use process-algebra style reasoning to verify specifications written in this style. I think the first part is fairly reasonable; the verification part is still pretty kludgy and needs a lot of work.

1 Specification

I first describe the notation in terms of a simple example. I then indicate the general notation and approximately what it means.

1.1 An Example

The example is a specification of a simple, single-user memory. The user sends either a $\langle \text{“}\mathsf{Read}\text{”}, l \rangle$ request to read location l or a $\langle \text{“}\mathsf{Write}\text{”}, l, v \rangle$ request to set location l to v . The memory responds to a read request with $\langle \text{“}\mathsf{OK}\text{”}, v \rangle$, where v is the current value of location l, and it responds to a write request with $\langle \text{“}\mathsf{OK}\text{”} \rangle$. No requests need ever occur, but the memory must eventually respond to every request.

The specification is a bit more complicated than necessary because it changes the memory with a separate, internal action. I did that to make the example a bit more interesting. I assume that the action $Send(v, c)$, which sends the value v over channel c, is already defined. I let *Locs* and *Vals* be the sets of possible memory locations and memory values, and *InitMem* be the set of possible initial memory values. I use the TLA notation in which $[x \text{ EXCEPT } ![i] = u]$ is array (function) $\widehat{x}$ that is the same as x except $\widehat{x}[i] = u$.

Here is the specification:

$$
\begin{array}{l}
\textbf{ProcAction } N(pc) \;\triangleq \\
\quad \Big(\bigoplus_{l \in Locs} \Big(\; Send(\langle \text{“Rd”}, l \rangle, c)_{mem}; \;\; rr : Send(\langle \text{“OK”}, mem[l] \rangle)_{mem} \\
\qquad\qquad\qquad \oplus \\
\qquad\qquad \bigoplus_{v \in Vals} Send(\langle \text{“Wr”}, l, v \rangle, c)_{mem} \,; \\
\qquad\qquad\qquad rw : \big(\; (mem' = [mem \text{ EXCEPT } ![l] = v])_c; \\
\qquad\qquad\qquad\qquad Send(\langle \text{“OK”} \rangle)_{mem} \;\; \big) \qquad\qquad \Big) \Big)^*
\end{array}
$$

$$
\begin{array}{l}
Spec \;\triangleq\; \exists\, mem, pc \; : \\
\qquad \land \; (mem' \in InitMem) \land At(N(pc)) \\
\qquad \land \; \Box[N(pc)]_{\langle mem, pc \rangle} \\
\qquad \land \; \forall\, l \in Locs \; : \; \land \; \mathrm{WF}_{pc}(rr(pc, l)) \\
\qquad\qquad\qquad\qquad\quad\;\; \land \; \forall\, v \in Vals \; : \; \mathrm{WF}_{pc}(rw(pc, l, v))
\end{array}
$$

The **ProcAction** command defines the action $N(pc)$ to equal the following, where x, y, and z are arbitrary constants, and the constants x, y, z, rr, and rw are assumed to be distinct.

$$
\begin{array}{l}
\exists\, l \in Locs \; : \\
\quad \lor \; \lor \; \land \; pc = x \\
\qquad\quad \land \; pc' = \langle rr, l \rangle \\
\qquad\quad \land \; Send(\langle \text{“Rd”}, l \rangle, c) \land (mem' = mem) \\
\qquad \lor \; \land \; pc = \langle rr, l \rangle \\
\qquad\quad \land \; pc' = x \\
\qquad\quad \land \; Send(\langle \text{“OK”}, mem[l] \rangle) \land (mem' = mem) \\
\quad \lor \; \exists\, v \in Vals \; : \\
\qquad \lor \; \land \; pc = x \\
\qquad\quad \land \; pc' = \langle rw, l, v \rangle \\
\qquad\quad \land \; Send(\langle \text{“Wr”}, l, v \rangle, c) \land (mem' = mem) \\
\qquad \lor \; \land \; pc = \langle rw, l, v \rangle \\
\qquad\quad \land \; pc' = \langle z, l, v \rangle \\
\qquad\quad \land \; (mem' = [mem \text{ EXCEPT } ![l] = v]) \land (c' = c) \\
\qquad \lor \; \land \; pc = \langle z, l, v \rangle \\
\qquad\quad \land \; pc' = x \\
\qquad\quad \land \; Send(\langle \text{“OK”} \rangle, c) \land (mem' = mem)
\end{array}
$$

It also makes the following definitions (among others):

$$
\begin{array}{lcl}
At(N(pc)) & \triangleq & pc = x \\
rr(pc, l) & \triangleq & \land\ pc = \langle rr, l \rangle \\
 & & \land\ pc' = x \\
 & & \land\ Send(\langle \text{“OK”}, mem[l] \rangle) \land (mem' = mem) \\
rw(pc, l, v) & \triangleq & \lor\ \land\ pc = \langle rw, l, v \rangle \\
 & & \quad \land\ pc' = \langle z, l, v \rangle \\
 & & \quad \land\ (mem' = [mem \text{ EXCEPT } ![l] = v]) \land (c' = c) \\
 & & \lor\ \land\ pc = \langle rw, l, v \rangle \\
 & & \quad \land\ pc' = x \\
 & & \quad \land\ Send(\langle \text{“OK”} \rangle, c) \land (mem' = mem)
\end{array}
$$

1.2 The General Notation

The right-hand side of a **ProcAction** statement is an expression formed by combining ordinary TLA actions with the following additional operators[1]

$$; \qquad \oplus \qquad \bigoplus \qquad (\ldots)^* \qquad \uparrow \qquad \| \qquad \Big\| $$

If A is an ordinary TLA action, then we let A_f be an abbreviation for $A \land (f' = f)$, allowing us to write UNCHANGED expressions more compactly. The additional operators have the following intuitive interpretation.

$A; B$ — Do A then B.

$A \oplus B$ — Do A or B.

$\bigoplus_{v \in S} A(v)$ — Do $A(v)$ for some $v \in S$.

$\bigoplus_{v} A(v)$ — Do $A(v)$ for some v.

$(A)^*$ — Keep doing A actions forever, or until the loop is exited (see below).

$A \uparrow$ — Do A, then exit from the innermost containing $(\ldots)^*$.

$A \| B$ — Interleave A and B. (If A and B are ordinary TLA actions, then $A \| B$ is equivalent, in a sense explained below, to $A; B \oplus B; A$.)

[1] I'm not completely convinced that $\|$ and $\Big\|$ are necessary.

$\mathop{\|}\limits_{v \in S} A(v)$ and $\mathop{\|}\limits_{v} A(v)$ — Interleave the $A(v)$ for all $v \in S$ and all v, respectively.

A label can be attached to any subexpression. (All labels must be unique within the **ProcAction** statement.) If the subexpression $l : A$ appears in a **ProcAction** statement, then the following actions and predicates are defined, where *pc* is the **ProcAction** variable and $v_1, \ldots, v_n$ is the sequence of bound $\bigoplus$ variables containing the subexpression.

$l(pc, v_1, \ldots, v_n)$ A TLA action. A step of this action consists of performing a step of one of the subactions of A.

$At(l(pc, v_1, \ldots, v_n))$ The predicate asserting that control is at the beginning of the subexpression.

$In(l(pc, v_1, \ldots, v_n))$ The predicate asserting that control is at the beginning or somewhere inside the subexpression.

$After(l(pc, v_1, \ldots, v_n))$ The predicate asserting that control is immediately after the subexpression.

It's fairly straightforward to give a formal semantics to the **ProcAction** statement with the operators I've defined so far. For future reference, I'll sketch how it's done.

Let a primitive action be an expression of the form $l : A$, where A is an ordinary TLA action. A p-action is an expression constructed from such primitive actions using the operators ";", $\oplus$, etc., where the labels l are all distinct. A **ProcAction** statement defines a p-action for the entire right-hand side, as well as for each label. Assume a control variable *pc*, distinct from all variables that appear in primitive actions. We will define a semantics of p-actions by defining, for each p-action P:

- A collection of constants L_P called *labels*, with a distinguished element at_P called the *at* label.
- A collection of actions A_P of the form $(pc = l) \land (pc' = k) \land A$, where A is an ordinary TLA action in which *pc* does not occur, $l \in L_P$, and k is either a label in L_P or one of the special constants "Done" or "Exit".

We can then define

$$\begin{array}{lcl} Act(P) & \triangleq & \exists A \in A_P : A \\ At(P) & \triangleq & pc = at_P \\ In(P) & \triangleq & pc \in L_P \\ After(P) & \triangleq & pc = \text{"Done"} \end{array}$$

These are the actions and predicates described informally above, where if P has the label l and $v_1, \ldots, v_n$ are the enclosing $\bigoplus$ variables, then we write $l(pc, v_1, \ldots, v_n)$ instead of $Act(P)$, $At(l(pc, v_1, \ldots, v_n))$ instead of $At(P)$, etc.

We then recursively define L_P, at_P, and A_P for any p-action P. Here are some of the recursive definitions:

- If P is the primitive action $l : A$, then $L_P \triangleq \{l\}$ and A_P contains the single action $(pc = l) \wedge (pc' = \text{“Done”}) \wedge A$.

- If $P = P1; P2$, then $L_P \triangleq L_{P1} \cup L_{P2}$; $at_P \triangleq at_{P1}$; and $A_P \triangleq \widehat{A_{P1}} \cup A_{P2}$, where $\widehat{A_{P1}}$ consists of the actions of A_{P1} with at_{P2} substituted for “Done”.

- If $P = \bigoplus_{v \in S} Q$, then L_P is the set consisting of at_Q together with all elements of the form $\langle v, l \rangle$ with $v \in S$ and $l \in L_Q$, $l \neq at_Q$; and A_P consists of the set of all actions obtained from actions in A_Q by replacing every label l in L_Q different from at_Q by $\langle v, l \rangle$, for all $v \in S$.

- If $P = P1 \| P2$, then L_P is the set of all labels $\langle l1, l2 \rangle$, where li is either in L_{Pi} or equals “Done”, excluding $\langle \text{“Done”}, \text{“Done”} \rangle$; $at_P \triangleq \langle at_{P1}, at_{P2} \rangle$; and $A_P \triangleq \widehat{A_{P1}} \cup \widehat{A_{P2}}$, where $\widehat{A_{P1}}$ consists of all actions of the form $(pc = \langle l, l2 \rangle) \wedge (pc' = \langle k, l2 \rangle) \wedge A$, for some $(pc = l) \wedge (pc' = k) \wedge A$ in A_{P1} and some $l2$ in L_{P2} or equal to ”Done”, except writing $pc' = \text{“Done”}$ instead of $pc' = \langle \text{“Done”}, \text{“Done”} \rangle$, and $pc' = \text{“Exit”}$ instead of $pc' = \langle \text{“Exit”}, l2 \rangle$.

The definitions for the other constructs are similar.

1.3 Is This Enough?

I don't see any need for additional operators. I think that the only missing standard CCS operators are hiding and more general recursion than simple looping. Hiding is expressed with the ordinary TLA quantifier $\boldsymbol{\exists}$. It shouldn't be hard to extend the language of p-actions to allow recursive definitions. For a recursively defined p-action P, the sets L_P and A_P become infinite, but that shouldn't cause any problem. However, I don't think this kind of recursive definition is necessary. I think that recursion should be restricted to the definitions of data types. For example, here's a specification of a bounded buffer, with input channel *in* and output channel *out*, in which a *Put* request waits if the buffer is full, and a *Get* request waits if the buffer is empty.

$$
\begin{array}{l}
\textbf{ProcAction } B(pc) \;\triangleq \\
\quad (\bigoplus_{v \in Vals} Send(v, in)_{\langle q, out \rangle} \,; \\
\qquad\qquad p \,:\, (Len(q) \neq Max) \land (q' = q \circ \langle v \rangle) \land Send(\text{“OK”}, in)_{out} \quad)^* \\
\quad \| \\
\quad (Send(\text{“Get”}, out)_{\langle q, in \rangle} \,; \\
\qquad g \,:\, (q \neq \langle\,\rangle) \land (q' = Tail(q)) \land Send(Head(q), out)_{in} \quad)^*
\end{array}
$$

$$
\begin{array}{l}
Spec \;\triangleq\; \exists\, q, pc \,:\, \land\, (q = \langle\,\rangle) \land At(B(pc)) \\
\qquad\qquad\qquad\quad \land\, \Box[B(pc)]_{\langle q, in, out, pc \rangle} \\
\qquad\qquad\qquad\quad \land\, WF_{pc}(g(pc)) \land \forall\, v \in Vals \,:\, WF_{pc}(p(pc, v))
\end{array}
$$

Here's an alternative way of writing the specification that does not use the $\|$ construct.

$$
\begin{array}{l}
\textbf{ProcAction } Put(pc) \;\triangleq \\
\quad (\bigoplus_{v \in Vals} Send(v, in)_{\langle q, out \rangle} \,; \\
\qquad\qquad p \,:\, (Len(q) \neq Max) \land (q' = q \circ \langle v \rangle) \land Send(\text{“OK”}, in)_{out} \,)^* \\
\textbf{ProcAction } Get(pc) \;\triangleq \\
\quad (Send(\text{“Get”}, out)_{\langle q, in \rangle} \,; \\
\qquad g \,:\, (q \neq \langle\,\rangle) \land (q' = Tail(q)) \land Send(Head(q), out)_{in} \,)^* \\
Spec \;\triangleq \\
\quad \exists\, q, pc1, pc2 \,: \\
\qquad \land\, (q = \langle\,\rangle) \land At(Put(pc1)) \land At(Get(pc2)) \\
\qquad \land\, \Box[Put(pc1)]_{in} \land \Box[Get(pc2)]_{out} \land \Box[(pc1' \neq pc1) \lor (pc2' \neq pc2)]_q \\
\qquad \land\, WF_{pc}(g(pc2)) \land \forall\, v \in Vals \,:\, WF_{pc}(p(pc1, v))
\end{array}
$$

One might think of adding a synchronous parallel composition operator $|||$, where $A|||B$ is equivalent to $A; B \oplus B; A \oplus A \land B$ for primitive actions A and B. However, I think that conjunction of temporal formulas can be used just as easily to express synchronous composition.

2 Verification

Those of you old enough to remember

```
AUTHOR = "Susan Owicki and Leslie Lamport",
```

```
TITLE = "Proving Liveness Properties of Concurrent
         Programs",
JOURNAL = toplas,
volume = 4,
number = 3,
YEAR = 1982,
month = JUL,
PAGES = "455--495"
```

will recognize the *At*, *In*, and *After* control predicates. What I've done is define a tiny programming language. I know from experience that these control predicates are all you need to prove properties about programs. However, I expect that we can simulate the process-algebra style of proof rules for reasoning about p-actions

Here's a quick sketch of my idea for how this is done. I'll stick to proving safety properties. We have to prove

$$\exists\, z \,:\, Init1 \wedge \Box[N1]_{\langle x,z\rangle} \Rightarrow \exists\, y \,:\, Init2 \wedge \Box[N2]_{\langle x,y\rangle}$$

For this, it suffices to find a refinement mapping—a function $\overline{y}$ of x and z—and prove $Init1 \Rightarrow \overline{Init2}$ and

$$[N1 \wedge I]_{\langle x,z\rangle} \Rightarrow [\overline{N2}]_{\langle x,\overline{y}\rangle} \tag{1}$$

where I is a suitable invariant and overbarring means substituting $\overline{y}$ for y. I'll ignore initial conditions and just consider (1). I'll suppose $N1$ and $N2$ are written as p-actions with control variables $pc1$ and $pc2$, respectively.[2] Then (1) can be written

$$[Act(N1) \wedge I]_{\langle x,z,pc1\rangle} \Rightarrow [\overline{Act(N2)}]_{\langle x,\overline{y},\overline{pc2}\rangle} \tag{2}$$

In general, $\overline{y}$ will be a function of x and z, and we'll we'll wind up defining $\overline{pc2}$ to be a function of $pc1$, so (2) reduces to

$$Act(N1) \wedge I \Rightarrow [\overline{Act(N2)}]_{\langle x,\overline{y},\overline{pc2}\rangle} \tag{3}$$

More precisely, we assume that we're given the refinement mapping $\overline{y}$ for the explicit internal variables y (excluding $pc2$), and we have to show that there exists a function $\overline{pc2}$ of $pc1$ so that (3) holds. The idea is to do this recursively for the subexpressions of $N1$ and $N2$.

[2] In general, $N1$ and $N2$ may just include p-actions as disjuncts. In this case, the type of verification I describe here is just part of the reasoning.

Let A and B be p-actions with control variables $pc1$ and $pc2$, respectively. Let $C \overset{A|B}{\underset{f|g}{\Longrightarrow}} D$ mean that there exists a mapping λ from L_A to $L_B \cup$ {"Done", "Exit"} with $\lambda(at_A) = at_B$ such that $(f' = f) \Rightarrow (g' = g)$ and

$$\begin{array}{l} \land\ \forall\, l \in L_A\ :\ \land\ (pc1 = l) \Rightarrow (pc2 = \lambda(l)) \\ \qquad\qquad\qquad \land\ (pc1' = l) \Rightarrow (pc2' = \lambda(l)) \\ \land\ (pc1' = \text{“Done”}) \Rightarrow (pc2' = \text{“Done”}) \\ \land\ (pc1' = \text{“Exit”}) \Rightarrow (pc2' = \text{“Exit”}) \\ \land\ C \\ \Rightarrow D \lor ((g' = g) \land (pc2' = pc2)) \end{array}$$

Let's now let $\overline{F}$ be the formula obtained by substituting $\overline{y}$ for y in F, and let $\overline{\overline{F}}$ be the formula obtained by substituting $\overline{pc2}$ for $pc2$ in $\overline{F}$. Let's also abbreviate $Act(A)$ to A. Then (3) becomes

$$N1 \land I \Rightarrow [\overline{\overline{N2}}]_{\langle x, \overline{y}, \overline{pc2} \rangle} \tag{4}$$

To prove that there exists $\overline{pc2}$ for which (4) holds, it suffices to prove

$$N1 \land I \overset{N1|N2}{\underset{\langle x,y \rangle | \langle x, \overline{y} \rangle}{\Longrightarrow}} \overline{N2}$$

We do this by applying "algebraic" rules to decompose the problem. First, there is a rule for primitive actions and for each operator—for example:

- If A and B are primitive actions, $(f' = f) \Rightarrow (g' = g)$, and $I \land A \Rightarrow [B]_g$, then $I \land A \overset{A|B}{\underset{f|g}{\Longrightarrow}} B$.
- If $A1 \overset{A1|B1}{\underset{f|g}{\Longrightarrow}} B1$ and $A2 \overset{A2|B2}{\underset{f|g}{\Longrightarrow}} B2$, then $A1; A2 \overset{A1;A2|B1;B2}{\underset{f|g}{\Longrightarrow}} B1; B2$.
- If $A \overset{A|B}{\underset{f|g}{\Longrightarrow}} B$ for all v, then $\bigoplus_v A \overset{\oplus_v A | \oplus_v B}{\underset{f|g}{\Longrightarrow}} \bigoplus_v B$

The relation $\overset{A|B}{\underset{f|g}{\Longrightarrow}}$ also obeys some general logical rules, such as:

- $I \Rightarrow (C \overset{A|B}{\underset{f|g}{\Longrightarrow}} D)$ iff $(I \land C) \overset{A|B}{\underset{f|g}{\Longrightarrow}} D$.
- Transitivity: $X \overset{A|B}{\underset{f|g}{\Longrightarrow}} Y$ and $Y \overset{B|C}{\underset{g|h}{\Longrightarrow}} Z$ imply $X \overset{A|C}{\underset{f|h}{\Longrightarrow}} Z$.

We can also define an equivalence relation $\underset{f}{\Leftrightarrow}$, where $A \underset{f}{\Leftrightarrow} B$ iff $A \overset{A|B}{\underset{f|f}{\Longrightarrow}} B$ and $B \overset{B|A}{\underset{f|f}{\Longrightarrow}} A$. For example, if A and B are primitive actions, then

$$A\|B \underset{f}{\Leftrightarrow} A; B \oplus B; A$$

for any f. Another useful equivalence is $(A; f' = f) \underset{f}{\Leftrightarrow} A$, which expresses stuttering equivalence. The relation $\underset{f}{\Leftrightarrow}$ should play the role of bisimulation equivalence in process algebra.