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 \( \langle \text{"Read"}, l \rangle \) request to read location \( l \) or a \( \langle \text{"Write"}, l, v \rangle \) request to set location \( l \) to \( v \). The memory responds to a read request with \( \langle \text{"OK"}, v \rangle \), where \( v \) is the current value of location \( l \), and it responds to a write request with \( \langle \text{"OK"} \rangle \). 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 \( \text{Send}(v, c) \), which sends the value \( v \) over channel \( c \), is already defined. I let \( \text{Locs} \) and \( \text{Vals} \) be the sets of possible memory locations and memory values, and \( \text{InitMem} \) be the set of possible initial memory values. I use the TLA notation in which \( [x \text{ EXCEPT } ![i] = u] \) is array (function) \( \hat{x} \) that is the same as \( x \) except \( \hat{x}[i] = u \).
Here is the specification:

**ProcAction** \( N(pc) \triangleq \)
\[
\bigoplus_{l \in \text{Locs}} (\text{Send}((\text{“Rd”}, l), c)_{\text{mem}}; \text{rr} : \text{Send}((\text{“OK”}, \text{mem}[l]))_{\text{mem}} \\
\bigoplus_{v \in \text{Vals}} \text{rw} : (\text{mem’} = [\text{mem EXCEPT ![l] = v}]_{c}; \text{Send}((\text{“OK”}))_{\text{mem}} )
\]

\[\text{Spec} \triangleq \exists \text{mem}, pc : \]
\[
\land (\text{mem’} \in \text{InitMem}) \land \text{At}(N(pc)) \\
\land \Box[N(pc)]_{(\text{mem}, pc)} \\
\land \forall l \in \text{Locs} : \land \text{WF}_{pc}(\text{rr}(pc, l)) \\
\land \forall v \in \text{Vals} : \land \text{WF}_{pc}(\text{rw}(pc, l, v))
\]

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

\[\exists l \in \text{Locs} : \]
\[
\lor \lor \land pc = x \\
\land pc' = \langle \text{rr}, l \rangle \\
\land \text{Send}(\langle \text{“Rd”}, l, c \rangle) \land (\text{mem’} = \text{mem}) \\
\lor \land pc = \langle \text{rr}, l \rangle \\
\land pc' = x \\
\land \text{Send}(\langle \text{“OK”}, \text{mem}[l] \rangle) \land (\text{mem’} = \text{mem})
\]

\[\lor \lor \exists v \in \text{Vals} : \]
\[
\lor \land pc = x \\
\land pc' = \langle \text{rw}, l, v \rangle \\
\land \text{Send}(\langle \text{“Wr”}, l, v, c \rangle) \land (\text{mem’} = \text{mem}) \\
\lor \land pc = \langle \text{rw}, l, v \rangle \\
\land pc' = \langle z, l, v \rangle \\
\land (\text{mem’} = [\text{mem EXCEPT ![l] = v}]_{c}) \land (c' = c) \\
\lor \land pc = \langle z, l, v \rangle \\
\land pc' = x \\
\land \text{Send}(\langle \text{“OK”}, c \rangle) \land (\text{mem’} = \text{mem})
\]
It also makes the following definitions (among others):

\[ At(N(pc)) \triangleq pc = x \]
\[ rr(pc, l) \triangleq pc = \langle rr, l \rangle \]
\[ pc' = x \]
\[ Send(\langle "OK", mem[l] \rangle) \land (\text{mem}' = \text{mem}) \]
\[ rw(pc, l, v) \triangleq \bot \land pc = \langle rw, l, v \rangle \]
\[ pc' = \langle z, l, v \rangle \]
\[ (\text{mem}' = [\text{mem} \setminus !l = v]) \land (c' = c) \]
\[ \vee \land pc = \langle rw, l, v \rangle \]
\[ pc' = x \]
\[ Send(\langle "OK" \rangle, c) \land (\text{mem}' = \text{mem}) \]

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\(^1\):

\[ \cdot ; \oplus \bigoplus (\ldots)^* \uparrow \| \| \]

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 \( \| \) are necessary.
∥ A(v) and ∥ A(v) — Interleave the A(v) for all v ∈ 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 v1, . . . , vn is the sequence of bound ⊕ variables containing the subexpression.

l(pc, v1, . . . , vn) A TLA action. A step of this action consists of performing a step of one of the subactions of A.

At(l(pc, v1, . . . , vn)) The predicate asserting that control is at the beginning of the subexpression.

In(l(pc, v1, . . . , vn)) The predicate asserting that control is at the beginning or somewhere inside the subexpression.

After(l(pc, v1, . . . , vn)) 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 “;”, ⊕, 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 LP called labels, with a distinguished element atP called the at label.

• A collection of actions AP of the form (pc = l) ∧ (pc′ = k) ∧ A, where A is an ordinary TLA action in which pc does not occur, l ∈ LP, and k is either a label in LP or one of the special constants “Done” or “Exit”.

We can then define

\[
\begin{align*}
\text{Act}(P) & \overset{\Delta}{=} \exists A \in AP : A \\
\text{At}(P) & \overset{\Delta}{=} pc = atP \\
\text{In}(P) & \overset{\Delta}{=} pc \in LP \\
\text{After}(P) & \overset{\Delta}{=} pc = \text{“Done”}
\end{align*}
\]
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 \( \text{Act}(P) \), \( \text{At}(l(pc, v_1, \ldots, v_n)) \) instead of \( \text{At}(P) \), etc.

We then recursively define \( L_P \), \( \text{at}_P \), 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 \((pc = l) \land (pc' = \text{"Done"}) \land A\).

- If \( P = P_1; P_2 \), then \( L_P \triangleq L_{P_1} \cup L_{P_2} \); \( \text{at}_P \triangleq \text{at}_{P_1} \); and \( A_P \triangleq \widetilde{A_{P_1}} \cup A_{P_2} \), where \( \widetilde{A_{P_1}} \) consists of the actions of \( A_{P_1} \) with \( \text{at}_{P_2} \) substituted for \("Done"\).

- If \( P = \bigoplus_{v \in S} Q \), then \( L_P \) is the set consisting of \( \text{at}_Q \) together with all elements of the form \((v, l)\) with \( v \in S \) and \( l \in L_Q \), \( l \neq \text{at}_Q \); 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 \( \text{at}_Q \) by \((v, l)\), for all \( v \in S \).

- If \( P = P_1 | P_2 \), then \( L_P \) is the set of all labels \( (l_1, l_2) \), where \( l_i \) is either in \( L_{P_1} \) or equals \("Done"\), excluding \( (\text{"Done"}, \text{"Done"}) \); \( \text{at}_P \triangleq (\text{at}_{P_1}, \text{at}_{P_2}) \); and \( A_P \triangleq \widetilde{A_{P_1}} \cup \widetilde{A_{P_2}} \), where \( \widetilde{A_{P_1}} \) consists of all actions of the form \((pc = (l_1, l_2)) \land (pc' = (k, l_2)) \land A\), for some \((pc = l) \land (pc' = k) \land A\) in \( A_{P_1} \) and some \( l_2 \) in \( L_{P_2} \) or equal to "Done", except writing \( pc' = \text{"Done"} \) instead of \( pc' = (\text{"Done"}, \text{"Done"}) \), and \( pc' = \text{"Exit"} \) instead of \( pc' = (\text{"Exit"}, l_2) \).

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 \( \exists \). 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 \( \text{in} \) and output channel \( \text{out} \), in
which a *Put* request waits if the buffer is full, and a *Get* request waits if the buffer is empty.

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

\[
\text{Spec } \triangleq \exists q, pc : (q = \langle \rangle) \land \text{At}(B(pc)) \\
\land [B(pc)]_{\langle q, in, out, pc \rangle} \\
\land \text{WF}_{pc}(g(pc)) \land \forall v \in \text{Vals} : \text{WF}_{pc}(p(pc, v))
\]

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

\[
\text{ProcAction } \text{Put}(pc) \triangleq \\
\begin{aligned}
( \bigoplus_{v \in \text{Vals}} & \text{Send}(v, in)_{\langle q, out \rangle} ; \\
\text{ } & p : (\text{Len}(q) \neq \text{Max}) \land (q' = q \circ \langle v \rangle) \land \text{Send(“OK”, in)}_{out} )^* 
\end{aligned}
\]

\[
\text{ProcAction } \text{Get}(pc) \triangleq \\
( \text{Send(“Get”, out)}_{\langle q, in \rangle} ; \\
\text{ } & g : (q \neq \langle \rangle) \land (q' = \text{Tail}(q)) \land \text{Send(Head}(q), out)_{in} )^* 
\]

\[
\text{Spec } \triangleq \exists q, pc1, pc2 : \\
\land (q = \langle \rangle) \land \text{At}(\text{Put}(pc1)) \land \text{At}(\text{Get}(pc2)) \\
\land \square[\text{Put}(pc1)]_{in} \land \square[\text{Get}(pc2)]_{out} \land \square[(pc1' \neq pc1) \lor (pc2' \neq pc2)]_q \\
\land \text{WF}_{pc}(g(pc2)) \land \forall v \in \text{Vals} : \text{WF}_{pc}(p(pc1, v))
\]

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",

6
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:

\[ \exists z : \text{Init}_1 \land [N_1]_{(x,z)} \Rightarrow \exists y : \text{Init}_2 \land [N_2]_{(x,y)} \]

For this, it suffices to find a refinement mapping—a function \( \overline{y} \) of \( x \) and \( z \)—and prove \( \text{Init}_1 \Rightarrow \text{Init}_2 \) and

\[ [N_1 \land I]_{(x,z)} \Rightarrow [N_2]_{(x,\overline{y})} \quad (1) \]

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 \( N_1 \) and \( N_2 \) are written as p-actions with control variables \( pc_1 \) and \( pc_2 \), respectively.\(^2\) Then (1) can be written

\[ [\text{Act}(N_1) \land I]_{(x,z,pc_1)} \Rightarrow [\text{Act}(N_2)]_{(x,\overline{y},pc_2)} \quad (2) \]

In general, \( \overline{y} \) will be a function of \( x \) and \( z \), and we’ll wind up defining \( pc_2 \) to be a function of \( pc_1 \), so (2) reduces to

\[ \text{Act}(N_1) \land I \Rightarrow [\text{Act}(N_2)]_{(x,\overline{y},pc_2)} \quad (3) \]

More precisely, we assume that we’re given the refinement mapping \( \overline{y} \) for the explicit internal variables \( y \) (excluding \( pc_2 \)), and we have to show that there exists a function \( pc_2 \) of \( pc_1 \) so that (3) holds. The idea is to do this recursively for the subexpressions of \( N_1 \) and \( N_2 \).

\(^2\) In general, \( N_1 \) and \( N_2 \) 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 $pc_1$ and $pc_2$, respectively. Let $C \xrightarrow{A|B} D$ mean that there exists a mapping $\lambda$ from $L_A$ to $L_B \cup \{\text{"Done"}, \text{"Exit"}\}$ with $\lambda(at_A) = at_B$ such that $(f' = f) \Rightarrow (g' = g)$ and

$$\land \forall l \in L_A : (pc_1 = l) \Rightarrow (pc_2 = \lambda(l))$$

$$\land (pc_1' = l) \Rightarrow (pc_2' = \lambda(l))$$

$$\land (pc_1' = \text{"Done"}) \Rightarrow (pc_2' = \text{"Done"})$$

$$\land (pc_1' = \text{"Exit"}) \Rightarrow (pc_2' = \text{"Exit"})$$

$$\land C$$

$$\Rightarrow D \lor ((g' = g) \land (pc_2' = pc_2))$$

Let’s now let $\overline{F}$ be the formula obtained by substituting $\overline{y}$ for $y$ in $F$, and let $\overline{F}'$ be the formula obtained by substituting $\overline{pc_2}$ for $pc_2$ in $\overline{F}$. Let’s also abbreviate $\text{Act}(A)$ to $A$. Then (3) becomes

$$N_1 \land I \Rightarrow \overline{N_2}_{(x, \overline{y}, \overline{pc_2})} \quad (4)$$

To prove that there exists $\overline{pc_2}$ for which (4) holds, it suffices to prove

$$N_1 \land I \xrightarrow{N_1|N_2} \overline{N_2}_{(x, \overline{y})}$$

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 $I \land A \xrightarrow{A|B} f|g} B$.

- If $A1 \xrightarrow{A1|B1} f|g} B1$ and $A2 \xrightarrow{A2|B2} f|g} B2$, then $A1; A2 \xrightarrow{A1;A2|B1;B2} f|g} B1; B2$.

- If $A \xrightarrow{A|B} f|g} B$ for all $v$, then $\bigoplus_v A \xrightarrow{\bigoplus_v A|\bigoplus_v B} f|g} \bigoplus_v B$.

The relation $\xrightarrow{A|B}$ also obeys some general logical rules, such as:

- $I \Rightarrow (C \xrightarrow{A|B} D) \text{ iff } (I \land C) \xrightarrow{A|B} f|g} D$.

- Transitivity: $X \xrightarrow{A|B} f|g} Y$ and $Y \xrightarrow{B|C} f|h} Z$ imply $X \xrightarrow{A|C} f|h} Z$. 8
We can also define an equivalence relation \( \leftrightarrow_f \), where \( A \leftrightarrow_f B \) iff \( A \xrightarrow{f\mid A} B \) and \( B \xrightarrow{f\mid A} A \). For example, if \( A \) and \( B \) are primitive actions, then

\[
A \parallel_f B \leftrightarrow_f A; B \oplus_f B; A
\]

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