```
──────────────────── MODULE CJupiter ────────────────────
Model of our own CJupiter protocol.
EXTENDS Integers, OT, TLC, AdditionalFunctionOperators, AdditionalSequenceOperators
─────────────────────────────────────────────────────────
CONSTANTS
    Client,      the set of client replicas
    Server,      the (unique) server replica
    Char,        set of characters allowed
    InitState    the initial state of each replica

Replica ≜ Client ∪ {Server}

List ≜ Seq(Char ∪ Range(InitState))             all possible lists/strings
MaxLen ≜ Cardinality(Char) + Len(InitState)     the max length of lists in any states;
    We assume that all inserted elements are unique.

ClientNum ≜ Cardinality(Client)
Priority ≜ CHOOSE f ∈ [Client → 1 . . ClientNum] : Injective(f)
─────────────────────────────────────────────────────────
ASSUME
    ∧ Range(InitState) ∩ Char = {}   due to the uniqueness requirement
    ∧ Priority ∈ [Client → 1 . . ClientNum]
─────────────────────────────────────────────────────────
The set of all operations. Note: The positions are indexed from 1.
Rd ≜ [type : {"Rd"}]
Del ≜ [type : {"Del"}, pos : 1 . . MaxLen]
Ins ≜ [type : {"Ins"}, pos : 1 . . (MaxLen + 1), ch : Char, pr : 1 . . ClientNum]   pr: priority

Op ≜ Ins ∪ Del
─────────────────────────────────────────────────────────
Cop: operation of type Op with context
Oid ≜ [c : Client, seq : Nat]   operation identifier
Cop ≜ [op : Op ∪ {Nop}, oid : Oid, ctx : SUBSET Oid]

tb: Is cop1 totally ordered before cop2?
This can be determined according to the serial view (sv) of any replica.
tb(cop1, cop2, sv) ≜
    LET pos1 ≜ FirstIndexOfElementSafe(sv, cop1.oid)
        pos2 ≜ FirstIndexOfElementSafe(sv, cop2.oid)
     IN  IF pos1 ≠ 0 ∧ pos2 ≠ 0  at the server or both are remote operations
          THEN pos1 < pos2       at a client: one is a remote operation and the other is a local operation
          ELSE pos1 ≠ 0
OT of two operations of type Cop.
COT(lcop, rcop) ≜ [lcop EXCEPT !.op = Xform(lcop.op, rcop.op), !.ctx = @ ∪ {rcop.oid}]
─────────────────────────────────────────────────────────
```

VARIABLES

For the client replicas:

$cseq,$ $cseq[c]$: local sequence number at client $c \in Client$

For all replicas: the n-ary ordered state space

$css,$ $css[r]$: the n-ary ordered state space at replica $r \in Replica$

$cur,$ $cur[r]$: the current node of css at replica $r \in Replica$

$state,$ $state[r]$: state (the list content) of replica $r \in Replica$

For edge ordering in CSS

$serial,$ $serial[r]$: the serial view of replica $r \in Replica$ about the server

$cincomingSerial,$

$sincomingSerial,$

For communication between the $Server$ and the Clients:

$cincoming,$ $cincoming[c]$: incoming channel at the client $c \in Client$

$sincoming,$ incoming channel at the $Server$

For model checking:

$chins$ a set of chars to insert

$serialVars \triangleq \langle serial, cincomingSerial, sincomingSerial \rangle$

$vars \triangleq \langle chins, cseq, css, cur, state, cincoming, sincoming, serialVars \rangle$

$comm \triangleq \text{INSTANCE } CSComm \text{ WITH } Msg \leftarrow Cop$

$commSerial \triangleq \text{INSTANCE } CSComm \text{ WITH } Msg \leftarrow Seq(Oid),$
$\quad cincoming \leftarrow cincomingSerial, sincoming \leftarrow sincomingSerial$

A css is a directed graph with labeled edges, represented by a record with node field and edge field. Each node is characterized by its context, a set of oids. Each edge is labeled with an operation.

$IsCSS(G) \triangleq$
$\quad \land G = [node \mapsto G.node, edge \mapsto G.edge]$
$\quad \land G.node \subseteq (\text{SUBSET } Oid)$
$\quad \land G.edge \subseteq [from : G.node, to : G.node, cop : Cop]$

$EmptySS \triangleq [node \mapsto \{\{\}\}, edge \mapsto \{\}]$

$TypeOK \triangleq$

For the client replicas:

$\quad \land cseq \in [Client \rightarrow Nat]$

For edge ordering in CSS:

$\quad \land serial \in [Replica \rightarrow Seq(Oid)]$
$\quad \land commSerial!TypeOK$

For all replicas: the n-ary ordered state space

$\quad \land \forall r \in Replica : IsCSS(css[r])$
$\quad \land cur \in [Replica \rightarrow \text{SUBSET } Oid]$
$\quad \land state \in [Replica \rightarrow List]$

For communication between the server and the clients:

```
∧ comm!TypeOK
```

For model checking:

```
∧ chins ⊆ Char
```

The *Init* predicate.

```
Init ≜
```

For the client replicas:

```
    ∧ cseq = [c ∈ Client ↦ 0]
```

For the server replica:

```
    ∧ serial = [r ∈ Replica ↦ ⟨⟩]
    ∧ commSerial!Init
```

For all replicas: the n-ary ordered state space

```
    ∧ css   = [r ∈ Replica ↦ EmptySS]
    ∧ cur = [r ∈ Replica ↦ {}]
    ∧ state = [r ∈ Replica ↦ InitState]
```

For communication between the server and the clients:

```
    ∧ comm!Init
```

For model checking:

```
    ∧ chins = Char
```

Locate the node in *rcss* (the *css* at replica $r \in Replica$) that matches the context *ctx* of cop.

```
Locate(cop, rcss) ≜ CHOOSE n ∈ rcss.node : n = cop.ctx
```

Take union of two state spaces *ss*1 and *ss*2.

```
ss1 ⊕ ss2 ≜ [node ↦ ss1.node ∪ ss2.node, edge ↦ ss1.edge ∪ ss2.edge]
```

xForm: Iteratively transform cop with a path through the *css* at replica $r \in Replica$, following the first edges.

```
xForm(cop, r) ≜
    LET rcss ≜ css[r]
        u ≜ Locate(cop, rcss)
        v ≜ u ∪ {cop.oid}
        RECURSIVE xFormHelper(_, _, _, _, _, _)
          'h' stands for "helper"; xcss: eXtra css created during transformation
        xFormHelper(uh, vh, coph, xcss, xcoph, xcurh) ≜
            IF uh = cur[r]
             THEN ⟨xcss, xcoph, xcurh⟩
             ELSE LET fedge ≜ CHOOSE e ∈ rcss.edge :
                                   ∧ e.from = uh
                                   ∧ ∀ uhe   ∈ rcss.edge :
                                       (uhe.from = uh ∧ uhe ≠ e) ⇒ tb(e.cop, uhe.cop, serial[r])
                      uprime ≜ fedge.to
                      fcop ≜ fedge.cop
```

```
                        coph2fcop ≜ COT(coph, fcop)
                        fcop2coph ≜ COT(fcop, coph)
                        vprime ≜ vh ∪ {fcop.oid}
                    IN  xFormHelper(uprime, vprime, coph2fcop,
                          [xcss EXCEPT !.node = @ ∪ {vprime},
                            !.edge = @ ∪ {[from ↦ vh, to ↦ vprime, cop ↦ fcop2coph],
                                          [from ↦ uprime, to ↦ vprime, cop ↦ coph2fcop]}],
                          coph2fcop, vprime)
  IN  xFormHelper(u, v, cop, [node ↦ {v}, edge ↦ {[from ↦ u, to ↦ v, cop ↦ cop]}], cop, v)
```

Perform cop at replica $r \in Replica$.

```
Perform(cop, r) ≜
   LET xform ≜ xForm(cop, r)   xform: ⟨xcss, xcop, xcur⟩
       xcss ≜ xform[1]
       xcop ≜ xform[2]
       xcur ≜ xform[3]
   IN  ∧ css′ = [css EXCEPT ![r] = @ ⊕ xcss]
       ∧ cur′ = [cur EXCEPT ![r] = xcur]
       ∧ state′ = [state EXCEPT ![r] = Apply(xcop.op, @)]
```

Client $c \in Client$ issues an operation op.

```
DoOp(c, op) ≜   op: the raw operation generated by the client c ∈ Client
     ∧ cseq′ = [cseq EXCEPT ![c] = @ + 1]
     ∧ LET cop ≜ [op ↦ op, oid ↦ [c ↦ c, seq ↦ cseq′[c]], ctx ↦ cur[c]]
       IN  ∧ Perform(cop, c)
           ∧ comm!CSend(cop)

DoIns(c) ≜
   ∃ ins ∈ {op ∈ Ins : op.pos ∈ 1 . . (Len(state[c]) + 1) ∧ op.ch ∈ chins ∧ op.pr = Priority[c]} :
     ∧ DoOp(c, ins)
     ∧ chins′ = chins \ {ins.ch}   We assume that all inserted elements are unique.
     ∧ UNCHANGED ⟨serialVars⟩

DoDel(c) ≜
   ∃ del ∈ {op ∈ Del : op.pos ∈ 1 . . Len(state[c])} :
     ∧ DoOp(c, del)
     ∧ UNCHANGED ⟨chins, serialVars⟩

Do(c) ≜
     ∨ DoIns(c)
     ∨ DoDel(c)
```

Client $c \in Client$ receives a message from the $Server$.

```
Rev(c) ≜
     ∧ comm!CRev(c)
     ∧ Perform(Head(cincoming[c]), c)
     ∧ commSerial!CRev(c)
```

$\quad \land serial' = [serial \text{ EXCEPT } ![c] = Head(cincomingSerial[c])]$
$\quad \land \text{UNCHANGED } \langle chins, cseq \rangle$

The *Server* receives a message.

$SRev \triangleq$
$\quad \land comm!SRev$
$\quad \land \text{LET } cop \triangleq Head(sincoming)$
$\quad\quad \text{IN } \quad \land Perform(cop, Server)$
$\quad\quad\quad\quad \land comm!SSendSame(cop.oid.c, cop)$ broadcast the original operation
$\quad\quad\quad\quad \land serial' = [serial \text{ EXCEPT } ![Server] = Append(@, cop.oid)]$
$\quad\quad\quad\quad \land commSerial!SSendSame(cop.oid.c, serial'[Server])$
$\quad \land \text{UNCHANGED } \langle chins, cseq, sincomingSerial \rangle$

The next-state relation.

$Next \triangleq$
$\quad \lor \exists c \in Client : Do(c) \lor Rev(c)$
$\quad \lor SRev$

The *Spec*. There is no requirement that the clients ever generate operations.

$Spec \triangleq Init \land \Box[Next]_{vars} \land \mathrm{WF}_{vars}(SRev \lor \exists c \in Client : Rev(c))$

The compactness of *CJupiter*: the *CSSes* at all replicas are the same.

$Compactness \triangleq$
$\quad comm!EmptyChannel \Rightarrow Cardinality(Range(css)) = 1$

THEOREM $Spec \Rightarrow Compactness$

---

* Modification History
* *Last* modified *Fri Nov* 16 14:47:14 *CST* 2018 by *hengxin*
* Created Sat *Sep* 01 11:08:00 *CST* 2018 by *hengxin*