```
──────────────────── 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 ∈ Client
    For all replicas: the n-ary ordered state space
    css,              css[r]: the n-ary ordered state space at replica r ∈ Replica
    cur,              cur[r]: the current node of css at replica r ∈ Replica
    state,            state[r]: state (the list content) of replica r ∈ Replica
    For edge ordering in CSS
    serial,           serial[r]: the serial view of replica r ∈ Replica about the server
    cincomingSerial,
    sincomingSerial,
    For communication between the Server and the Clients:
    cincoming,        cincoming[c]: incoming channel at the client c ∈ Client
    sincoming,        incoming channel at the Server
    For model checking:
    chins             a set of chars to insert
├──────────────────────────────────────────────────────────────────────────┤
serialVars ≜ ⟨serial, cincomingSerial, sincomingSerial⟩
vars ≜ ⟨chins, cseq, css, cur, state, cincoming, sincoming, serialVars⟩
├──────────────────────────────────────────────────────────────────────────┤
comm ≜ INSTANCE CSComm WITH Msg ← Cop
commSerial ≜ INSTANCE CSComm WITH Msg ← Seq(Oid),
                       cincoming ← cincomingSerial, sincoming ← 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) ≜
    ∧ G = [node ↦ G.node, edge ↦ G.edge]
    ∧ G.node ⊆ (SUBSET Oid)
    ∧ G.edge ⊆ [from : G.node, to : G.node, cop : Cop]

TypeOK ≜
    For the client replicas:
    ∧ cseq ∈ [Client → Nat]
    For edge ordering in CSS:
    ∧ serial ∈ [Replica → Seq(Oid)]
    ∧ commSerial!TypeOK
    For all replicas: the n-ary ordered state space
    ∧ ∀ r ∈ Replica : IsCSS(css[r])
    ∧ cur ∈ [Replica → SUBSET Oid]
    ∧ state ∈ [Replica → 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 ↦ [node ↦ {{}}, edge ↦ {}]]
    ∧ 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 ∈ Replica) that matches the context ctx of cop.
Locate(cop, rcss) ≜ CHOOSE n ∈ rcss.node : n = cop.ctx
Take union of two state spaces ss1 and ss2.
ss1 ⊕ ss2 ≜
    [ss1 EXCEPT !.node = @ ∪ ss2.node,
                !.edge  = @ ∪ ss2.edge]
xForm: Iteratively transform cop with a path through the css at replica r ∈ 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 ∈ Replica.

Perform(cop, r) ≜
    LET xform ≜ xForm(cop, r)
        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 ∈ 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 ∈ Client receives a message from the Server.

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

```
      ∧ serial′ = [serial EXCEPT ![c] = Head(cincomingSerial[c])]
      ∧ UNCHANGED ⟨chins, cseq⟩
--------------------------------------------------------------------------------
The Server receives a message.
SRev ≜
    ∧ comm!SRev
    ∧ LET cop ≜ Head(sincoming)
        IN    ∧ Perform(cop, Server)
              ∧ comm!SSendSame(cop.oid.c, cop)   broadcast the original operation
              ∧ serial′ = [serial EXCEPT ![Server] = Append(@, cop.oid)]
              ∧ commSerial!SSendSame(cop.oid.c, serial′[Server])
    ∧ UNCHANGED ⟨chins, cseq, sincomingSerial⟩
--------------------------------------------------------------------------------
The next-state relation.
Next ≜
    ∨ ∃ c ∈ Client : Do(c) ∨ Rev(c)
    ∨ SRev
The Spec. There is no requirement that the clients ever generate operations.
Spec ≜ Init ∧ □[Next]_vars ∧ WF_vars(SRev ∨ ∃ c ∈ Client : Rev(c))
--------------------------------------------------------------------------------
The compactness of CJupiter: the CSSes at all replicas are the same.
Compactness ≜
    comm!EmptyChannel ⇒ Cardinality(Range(css)) = 1

THEOREM Spec ⇒ Compactness
================================================================================
\* Modification History
\* Last modified Fri Nov 16 12:52:13 CST 2018 by hengxin
\* Created Sat Sep 01 11:08:00 CST 2018 by hengxin
```