```
1  ---------- MODULE CJupiter ----------
   Model of our own CJupiter protocol.

6  EXTENDS Integers, OT, TLC, AdditionalFunctionOperators, AdditionalSequenceOperators
7  ----------
8  CONSTANTS
9      Client,      the set of client replicas
10     Server,      the (unique) server replica
11     Char,        set of characters allowed
12     InitState    the initial state of each replica

14 Replica ≜ Client ∪ {Server}

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

20 ClientNum ≜ Cardinality(Client)
21 Priority ≜ CHOOSE f ∈ [Client → 1 .. ClientNum] : Injective(f)
22 ----------
23 ASSUME
24     ∧ Range(InitState) ∩ Char = {}
25     ∧ Priority ∈ [Client → 1 .. ClientNum]
26 ----------
   The set of all operations. Note: The positions are indexed from 1.
31 Rd ≜ [type : {"Rd"}]
32 Del ≜ [type : {"Del"}, pos : 1 .. MaxLen]
33 Ins ≜ [type : {"Ins"}, pos : 1 .. (MaxLen + 1), ch : Char, pr : 1 .. ClientNum]  pr: priority

35 Op ≜ Ins ∪ Del
36 ----------
   Cop: operation of type Op with context
40 Oid ≜ [c : Client, seq : Nat]  operation identifier
41 Cop ≜ [op : Op ∪ {Nop}, oid : Oid, ctx : SUBSET Oid, sctx : SUBSET Oid]

   tb: Is cop1 totally ordered before cop2?

   At a given replica r ∈ Replica, these can be determined in terms of sctx.
48 tb(cop1, cop2, r) ≜
49     ∨ cop1.oid ∈ cop2.sctx
50     ∨ ∧ cop1.oid ∉ cop2.sctx
51       ∧ cop2.oid ∉ cop1.sctx
52       ∧ cop1.oid.c ≠ r

   OT of two operations of type Cop.
57 COT(lcop, rcop) ≜
58     [op ↦ Xform(lcop.op, rcop.op), oid ↦ lcop.oid,
```

$ctx \mapsto lcop.ctx \cup \{rcop.oid\},\ sctx \mapsto lcop.sctx]$

VARIABLES

For the client replicas:

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

For the server replica:

$soids$, the set of operations the *Server* has executed

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 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

$comm \triangleq$ INSTANCE $CSComm$ WITH $Msg \leftarrow Cop$

$eVars \triangleq \langle chins \rangle$ variables for the environment

$cVars \triangleq \langle cseq \rangle$ variables for the clients

$ecVars \triangleq \langle eVars,\ cVars \rangle$ variables for the clients and the environment

$sVars \triangleq \langle soids \rangle$ variables for the server

$dsVars \triangleq \langle css,\ cur,\ state \rangle$ variables for the data structure: the n-ary ordered state space

$commVars \triangleq \langle cincoming,\ sincoming \rangle$ variables for communication

$vars \triangleq \langle eVars,\ cVars,\ sVars,\ commVars,\ dsVars \rangle$ all variables

---

An css is a directed graph with labeled edges.

It is represented by a record with node field and edge field.

Each node is characterized by its context, a set of operations.

Each edge is labeled with an operation. For clarity, we denote edges by records instead of tuples.

$IsCSS(G) \triangleq$

$\land\ G = [node \mapsto G.node,\ edge \mapsto G.edge]$

$\land\ G.node \subseteq (\text{SUBSET } Oid)$

$\land\ G.edge \subseteq [from : G.node,\ to : G.node,\ cop : Cop]$

$TypeOK \triangleq$

For the client replicas:

$\land\ cseq \in [Client \to Nat]$

For the server replica:

$\land\ soids \subseteq Oid$

```
        For all replicas: the n-ary ordered state space
124     ∧ ∀ r ∈ Replica : IsCSS(css[r])
125     ∧ cur ∈ [Replica → SUBSET Oid]
126     ∧ state ∈ [Replica → List]
        For communication between the server and the clients:
130     ∧ comm ! TypeOK
        For model checking:
134     ∧ chins ⊆ Char
135 ├──────────────────────────────────────────────────────────────────────┤
    The Init predicate.
139 Init ≜
140     ∧ chins = Char
        For the client replicas:
144     ∧ cseq = [c ∈ Client ↦ 0]
        For the server replica:
148     ∧ soids = {}
        For all replicas: the n-ary ordered state space
152     ∧ css  = [r ∈ Replica ↦ [node ↦ {{}}, edge ↦ {}]]
153     ∧ cur = [r ∈ Replica ↦ {}]
154     ∧ state = [r ∈ Replica ↦ InitState]
        For communication between the server and the clients:
158     ∧ comm ! Init
159 ├──────────────────────────────────────────────────────────────────────┤
    Client c ∈ Client issues an operation op.
163 DoOp(c, op) ≜   op: the raw operation generated by the client c ∈ Client
164         ∧ state′ = [state EXCEPT ![c] = Apply(op, @)]
165         ∧ cseq′ = [cseq EXCEPT ![c] = @ + 1]
166         ∧ LET cop ≜ [op ↦ op, oid ↦ [c ↦ c, seq ↦ cseq′[c]],
167               ctx ↦ cur[c], sctx ↦ {}]      cop: original operation with context
168                   v ≜ cur[c] ∪ {cop.oid}
169           IN    ∧ css′ = [css EXCEPT ![c].node = @ ∪ {v},
170                                      ![c].edge = @ ∪ {[from ↦ cur[c], to ↦ v, cop ↦ cop]}]
171                 ∧ cur′ = [cur EXCEPT ![c] = v]
172                 ∧ comm ! CSend(cop)

174 DoIns(c) ≜
175     ∃ ins ∈ Ins :
176         ∧ ins.pos ∈ 1 .. (Len(state[c]) + 1)
177         ∧ ins.ch ∈ chins
178         ∧ ins.pr = Priority[c]
179         ∧ chins′ = chins \ {ins.ch}   We assume that all inserted elements are unique.
180         ∧ DoOp(c, ins)
181         ∧ UNCHANGED sVars
```

```
183 DoDel(c) ≜
184     ∃ del ∈ Del :
185         ∧ del.pos ∈ 1 .. Len(state[c])
186         ∧ DoOp(c, del)
187         ∧ UNCHANGED ⟨sVars, eVars⟩

189 Do(c) ≜
190         ∨ DoIns(c)
191         ∨ DoDel(c)
```

Locate the node in *rcss* which matches the context *ctx* of cop.

rcss: the *css* at replica $r \in Replica$

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

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

```
203 xForm(cop, r) ≜
204     LET rcss ≜ css[r]
205         u ≜ Locate(cop, rcss)
206         v ≜ u ∪ {cop.oid}
207         RECURSIVE xFormHelper(_, _, _, _)
208          'h' stands for "helper"; xcss: eXtra css created during transformation
209         xFormHelper(uh, vh, coph, xcss) ≜
210             IF uh = cur[r]
211              THEN xcss
212              ELSE LET fedge ≜ CHOOSE e ∈ rcss.edge :
213                                 ∧ e.from = uh
214                                 ∧ ∀ uhe ∈ rcss.edge :
215                                     (uhe.from = uh ∧ uhe ≠ e) ⇒ tb(e.cop, uhe.cop, r)
216                       uprime ≜ fedge.to
217                       fcop ≜ fedge.cop
218                       coph2fcop ≜ COT(coph, fcop)
219                       fcop2coph ≜ COT(fcop, coph)
220                        vprime ≜ vh ∪ {fcop.oid}
221                   IN   xFormHelper(uprime, vprime, coph2fcop,
222                          [xcss EXCEPT !.node = @ ∘ ⟨vprime⟩,
223                                        the order of recording edges here is important
224                                       !.edge = @ ∘ ⟨[from ↦ vh, to ↦ vprime, cop ↦ fcop2coph],
225                                                     [from ↦ uprime, to ↦ vprime, cop ↦ coph2fcop]⟩])
226     IN   xFormHelper(u, v, cop, [node ↦ ⟨v⟩, edge ↦ ⟨[from ↦ u, to ↦ v, cop ↦ cop]⟩])
```

The *eXtra css* (*xcss*) updates the status of replica $r \in Replica$.

```
231 r ⊕ xcss ≜
232     LET xn ≜ xcss.node
233         xe ≜ xcss.edge
234         xcur ≜ Last(xn)
```

```
235             xcop ≜ Last(xe).cop
236     IN    ∧ css′ = [css EXCEPT ![r].node = @ ∪ Range(xn),
237                                  ![r].edge = @ ∪ Range(xe)]
238           ∧ cur′ = [cur EXCEPT ![r] = xcur]
239           ∧ state′ = [state EXCEPT ![r] = Apply(xcop.op, @)]
```

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

```
244 Rev(c) ≜
245       ∧ comm!CRev(c)
246       ∧ LET cop ≜ Head(cincoming[c])   the received original operation
247             xcss ≜ xForm(cop, c)       the eXtra part of css
248         IN  ∧ c ⊕ xcss
249       ∧ UNCHANGED ⟨ecVars, sVars⟩
```

The *Server* receives a message.

```
254 SRev ≜
255     ∧ comm!SRev
256     ∧ LET org ≜ Head(sincoming)   the received operation
257           cop ≜ [org EXCEPT !.sctx = soids]   set its sctx field
258           xcss ≜ xForm(cop, Server)   the eXtra part of css
259       IN  ∧ soids′ = soids ∪ {cop.oid}
260           ∧ Server ⊕ xcss
261           ∧ comm!SSendSame(cop.oid.c, cop)   broadcast the original operation
262     ∧ UNCHANGED ecVars
```

The next-state relation.

```
267 Next ≜
268     ∨ ∃ c ∈ Client : Do(c) ∨ Rev(c)
269     ∨ SRev
```

The *Spec*. (*TODO*: Check the fairness condition.)

```
273 Spec ≜ Init ∧ □[Next]_vars ∧ WF_vars(Next)
274
```

```
\* Modification History
\* Last modified Wed Sep 05 19:53:11 CST 2018 by hengxin
\* Created Sat Sep 01 11:08:00 CST 2018 by hengxin
```