```
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 ----------
41 Oid ≜ [c : Client, seq : Nat]  operation identifier
42 Cop ≜ [op : Op ∪ {Nop}, oid : Oid, ctx : SUBSET Oid, sctx : SUBSET Oid]  operation with context

44 cop1 ≺ cop2 ≜
45     ∨ cop2.sctx = {}
46     ∨ cop1.oid ∈ cop2.sctx

48 COT(lcop, rcop) ≜
49     [op ↦ Xform(lcop.op, rcop.op), oid ↦ lcop.oid,
50         ctx ↦ lcop.ctx ∪ {rcop.oid}, sctx ↦ lcop.sctx]
51 ----------
52 VARIABLES
       For the client replicas:
56     cseq,     cseq[c]: local sequence number at client c ∈ Client
57     cstate,   cstate[c]: state (the list content) of the client c ∈ Client
```

For the server replica:

61 $soids$, the set of operations the *Server* has executed

62 $sstate$, $sstate$: state (the list content) of the server *Server*

For all replicas: the n-ary ordered state space

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

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

For communication between the *Server* and the Clients:

71 $cincoming$, $cincoming[c]$: incoming channel at the client $c \in Client$

72 $sincoming$, incoming channel at the *Server*

For model checking:

76 $chins$ a set of chars to insert

78 ----

79 $comm \triangleq$ INSTANCE $CSComm$ WITH $Msg \leftarrow Cop$

80 ----

81 $eVars \triangleq \langle chins \rangle$ variables for the environment

82 $cVars \triangleq \langle cseq, cstate \rangle$ variables for the clients

83 $ecVars \triangleq \langle eVars, cVars \rangle$ variables for the clients and the environment

84 $sVars \triangleq \langle soids, sstate \rangle$ variables for the server

85 $dsVars \triangleq \langle css, cur \rangle$ variables for the data structure: the n-ary ordered state space

86 $commVars \triangleq \langle cincoming, sincoming \rangle$ variables for communication

87 $vars \triangleq \langle eVars, cVars, sVars, commVars, dsVars \rangle$ all variables

88 ----

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.

99 $IsCSS(G) \triangleq$

100 $\quad \land G = [node \mapsto G.node, edge \mapsto G.edge]$

101 $\quad \land G.node \subseteq (\text{SUBSET } Oid)$

102 $\quad \land G.edge \subseteq [from : G.node, to : G.node, cop : Cop]$

104 $TypeOK \triangleq$

For the client replicas:

108 $\quad \land cseq \in [Client \to Nat]$

109 $\quad \land cstate \in [Client \to List]$

For the server replica:

113 $\quad \land soids \subseteq Oid$

114 $\quad \land sstate \in List$

For all replicas: the n-ary ordered state space

118 $\quad \land \forall r \in Replica : IsCSS(css[r])$

119 $\quad \land cur \in [Replica \to \text{SUBSET } Oid]$

```
        For communication between the server and the clients:
123     ∧ comm ! TypeOK
        For model checking:
127     ∧ chins ⊆ Char
128 ├──────────────────────────────────────────────────────────────────────┤
    The Init predicate.
132 Init ≜
133     ∧ chins = Char
        For the client replicas:
137     ∧ cseq = [c ∈ Client ↦ 0]
138     ∧ cstate = [c ∈ Client ↦ InitState]
        For the server replica:
142     ∧ soids = {}
143     ∧ sstate = InitState
        For all replicas: the n-ary ordered state space
147     ∧ css = [r ∈ Replica ↦ [node ↦ {{}}, edge ↦ {}]]
148     ∧ cur = [r ∈ Replica ↦ {}]
        For communication between the server and the clients:
152     ∧ comm ! Init
153 ├──────────────────────────────────────────────────────────────────────┤
    Client c ∈ Client issues an operation op.
157 DoOp(c, op) ≜   op: the raw operation generated by the client c ∈ Client
158         ∧ cstate' = [cstate EXCEPT ![c] = Apply(op, @)]
159         ∧ cseq' = [cseq EXCEPT ![c] = @ + 1]
160         ∧ LET cop ≜ [op ↦ op, oid ↦ [c ↦ c, seq ↦ cseq'[c]],
161               ctx ↦ cur[c], sctx ↦ {}]      cop: original operation with context
162               v ≜ cur[c] ∪ {cop.oid}
163           IN  ∧ css' = [css EXCEPT ![c].node = @ ∪ {v},
164                                    ![c].edge = @ ∪ {[from ↦ cur[c], to ↦ v, cop ↦ cop]}]
165               ∧ cur' = [cur EXCEPT ![c] = v]
166               ∧ comm ! CSend(cop)

168 DoIns(c) ≜
169     ∃ ins ∈ Ins :
170         ∧ ins.pos ∈ 1 .. (Len(cstate[c]) + 1)
171         ∧ ins.ch ∈ chins
172         ∧ ins.pr = Priority[c]
173         ∧ chins' = chins \ {ins.ch}   We assume that all inserted elements are unique.
174         ∧ DoOp(c, ins)
175         ∧ UNCHANGED sVars

177 DoDel(c) ≜
178     ∃ del ∈ Del :
179         ∧ del.pos ∈ 1 .. Len(cstate[c])
```

```
180         ∧ DoOp(c, del)
181         ∧ UNCHANGED ⟨sVars, eVars⟩

183 Do(c) ≜
184       ∨ DoIns(c)
185       ∨ DoDel(c)
```

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

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

```
191 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.

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

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

```
225 r ⊕ xcss ≜
226     LET xn ≜ xcss.node
227         xe ≜ xcss.edge
228         xcur ≜ Last(xn)
229         xcop ≜ Last(xe).cop
230     IN  ∧ css′ = [css EXCEPT ![r].node = @ ∪ Range(xn),
231                              ![r].edge = @ ∪ Range(xe)]
```

```
232             ∧ cur′ = [cur EXCEPT ![r] = xcur]
233         ∧ cstate′ = [cstate EXCEPT ![r] = Apply(xcop.op, @)]

    Client c ∈ Client receives a message from the Server.
238 Rev(c) ≜
239     ∧ comm!CRev(c)
240     ∧ LET cop ≜ Head(cincoming[c])   the received original operation
241           xcss ≜ xForm(cop, c)       the eXtra part of css
242       IN  ∧ c ⊕ xcss
243           ∧ cstate′ = [cstate EXCEPT ![c] = Apply(Last(xcss.edge).cop.op, @)]
244     ∧ UNCHANGED ⟨cseq, sVars, eVars⟩
245 ──────────────────────────────────────────────────────────────────────

    The Server receives a message.
249 SRev ≜
250     ∧ comm!SRev
251     ∧ LET org ≜ Head(sincoming)      the received operation
252           cop ≜ [org EXCEPT !.sctx = soids]   set its sctx field
253           xcss ≜ xForm(cop, Server)   the eXtra part of css
254       IN  ∧ soids′ = soids ∪ {cop.oid}
255           ∧ Server ⊕ xcss
256           ∧ sstate′ = Apply(Last(xcss.edge).cop.op, sstate)   apply the transformed operation
257           ∧ comm!SSendSame(cop.oid.c, cop)   broadcast the original operation
258     ∧ UNCHANGED ecVars
259 ──────────────────────────────────────────────────────────────────────

    The next-state relation.
263 Next ≜
264     ∨ ∃ c ∈ Client : Do(c) ∨ Rev(c)
265     ∨ SRev
    The Spec. (TODO: Check the fairness condition.)
269 Spec ≜ Init ∧ □[Next]_vars ∧ WF_vars(Next)
270 ──────────────────────────────────────────────────────────────────────

    \* Modification History
    \* Last modified Tue Sep 04 23:22:46 CST 2018 by hengxin
    \* Created Sat Sep 01 11:08:00 CST 2018 by hengxin
```