```
1  ---------- MODULE XJupiterExtended ----------
   XJupiter extended with Cop with the sctx field.
5  EXTENDS Integers, OT, TLCUtils, AdditionalFunctionOperators, AdditionalSequenceOperators
6  ----------
7  CONSTANTS
8      Client,       the set of client replicas
9      Server,       the (unique) server replica
10     Char,         set of characters allowed
11     InitState     the initial state of each replica

13 Replica ≜ Client ∪ {Server}

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

19 ClientNum ≜ Cardinality(Client)
20 Priority ≜ CHOOSE f ∈ [Client → 1 .. ClientNum] : Injective(f)

22  direction flags
23 Local ≜ 0
24 Remote ≜ 1
25 ----------
26 ASSUME
27     ∧ Range(InitState) ∩ Char = {}   due to the uniqueness requirement
28     ∧ Priority ∈ [Client → 1 .. ClientNum]
29 ----------
   The set of all operations. Note: The positions are indexed from 1.
34 Rd ≜ [type : {"Rd"}]
35 Del ≜ [type : {"Del"}, pos : 1 .. MaxLen]
36 Ins ≜ [type : {"Ins"}, pos : 1 .. (MaxLen + 1), ch : Char, pr : 1 .. ClientNum]  pr: priority

38 Op ≜ Ins ∪ Del
39 ----------
   Cop: operation of type Op with context
43 Oid ≜ [c : Client, seq : Nat]   operation identifier
   Cop with the sctx field (the extended part)
47 Cop ≜ [op : Op ∪ {Nop}, oid : Oid, ctx : SUBSET Oid, sctx : SUBSET Oid]

   OT of two operations of type Cop.
52 COT(lcop, rcop) ≜ [lcop EXCEPT !.op = Xform(lcop.op, rcop.op), !.ctx = @ ∪ {rcop.oid}]
53 ----------
54 VARIABLES
       For the client replicas:
58     cseq,      cseq[c]: local sequence number at client c ∈ Client
```

For the server replica (the extended part):

62 $soids$, the set of operations the *Server* has executed

The $2D$ state spaces (ss, for short). Each client maintains one $2D$ state space. The server maintains n $2D$ state spaces, one for each client.

68 css, $css[c]$: the $2D$ state space at client $c \in Client$

69 $ccur$, $cur[c]$: the current node of $css[c]$

70 sss, $sss[c]$: the $2D$ state space maintained by the *Server* for client $c \in Client$

71 $scur$, $scur[c]$: the current node of $sss[c]$

For all replicas

75 $state$, $state[r]$: state (the list content) of replica $r \in Replica$

For communication between the *Server* and the Clients:

79 $cincoming$, $cincoming[c]$: incoming channel at the client $c \in Client$

80 $sincoming$, incoming channel at the *Server*

For model checking:

84 $chins$ a set of chars to insert

85 ----

86 $comm \triangleq$ INSTANCE $CSComm$ WITH $Msg \leftarrow Cop$

87 ----

88 $eVars \triangleq \langle chins \rangle$ variables for the environment

89 $cVars \triangleq \langle cseq \rangle$ variables for the clients

90 $cssVars \triangleq \langle css, ccur \rangle$ variables for $2D$ state spaces at clients

91 $sssVars \triangleq \langle sss, scur \rangle$ variables for $2D$ state spaces at the *Server*

92 $commVars \triangleq \langle cincoming, sincoming \rangle$ variables for communication

93 $vars \triangleq \langle eVars, cVars, commVars, cssVars, sssVars, state \rangle$ all variables

94 ----

A $2D$ state space 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 and a direction flag indicating whether this edge is LOCAL or REMOTE. For clarity, we denote edges by records instead of tuples.

103 $IsSS(G) \triangleq$

104 $\land\ G = [node \mapsto G.node,\ edge \mapsto G.edge]$

105 $\land\ G.node \subseteq (\text{SUBSET } Oid)$

106 $\land\ G.edge \subseteq [from : G.node,\ to : G.node,\ cop : Cop,\ lr : \{Local,\ Remote\}]$

108 $TypeOK \triangleq$

For the client replicas:

112 $\land\ cseq \in [Client \to Nat]$

For the $2D$ state spaces:

116 $\land\ \forall\, c \in Client : IsSS(css[c]) \land IsSS(sss[c])$

117 $\land\ ccur \in [Client \to \text{SUBSET } Oid]$

118 $\land\ scur \in [Client \to \text{SUBSET } Oid]$

119 $\land\ state \in [Replica \to List]$

For communication between the server and the clients:

```
123     ∧ comm ! TypeOK
        For model checking:
127     ∧ chins ⊆ Char
128 ├──────────────────────────────────────────────────────────────┤
    The Init predicate.
132 Init ≜
        For the client replicas:
136     ∧ cseq = [c ∈ Client ↦ 0]
        For the Server replica (the extended part):
140     ∧ soids = {}
        For the 2D state spaces:
144     ∧ css = [c ∈ Client ↦ [node ↦ {{}}, edge ↦ {}]]
145     ∧ ccur = [c ∈ Client ↦ {}]
146     ∧ sss = [c ∈ Client ↦ [node ↦ {{}}, edge ↦ {}]]
147     ∧ scur = [c ∈ Client ↦ {}]
        For all replicas:
151     ∧ state = [r ∈ Replica ↦ InitState]
        For communication between the server and the clients:
155     ∧ comm ! Init
        For model checking:
159     ∧ chins = Char
160 ├──────────────────────────────────────────────────────────────┤
    Locate the node in the 2D state space ss which matches the context ctx of cop.
164 Locate(cop, ss) ≜ CHOOSE n ∈ (ss.node) : n = cop.ctx

    xForm: iteratively transform cop with a path through the 2D state space ss at some client,
    following the edges with the direction flag d.
171 xForm(cop, ss, cur, d) ≜
172     LET u ≜ Locate(cop, ss)
173         v ≜ u ∪ {cop.oid}
174         RECURSIVE xFormHelper(_, _, _, _)
175          'h' stands for "helper"; xss: eXtra ss created during transformation
176         xFormHelper(uh, vh, coph, xss) ≜
177             IF uh = cur
178              THEN xss
179              ELSE LET e ≜ CHOOSE e ∈ ss.edge : e.from = uh ∧ e.lr = d
180                       uprime ≜ e.to
181                       copprime ≜ e.cop
182                       coph2copprime ≜ COT(coph, copprime)
183                       copprime2coph ≜ COT(copprime, coph)
184                        vprime ≜ vh ∪ {copprime.oid}
185                   IN   xFormHelper(uprime, vprime, coph2copprime,
186                            [xss EXCEPT !.node = @ ∘ ⟨vprime⟩],
```

```
                                        the order of recording edges here is important
                                        so that the last one is labeled with the final transformed operation
                                    !.edge = @ ∘ ⟨[from ↦ vh, to ↦ vprime, cop ↦ copprime2coph, lr ↦
                                                  [from ↦ uprime, to ↦ vprime, cop ↦ coph2copprime,
    IN  xFormHelper(u, v, cop, [node ↦ ⟨v⟩,
                                edge ↦ ⟨[from ↦ u, to ↦ v, cop ↦ cop, lr ↦ 1 − d]⟩])
```

Client $c \in Client$ perform operation cop guided by the direction flag d.

```
ClientPerform(cop, c, d) ≜
    LET xss ≜ xForm(cop, css[c], ccur[c], d)
        xn ≜ xss.node
        xe ≜ xss.edge
        xcur ≜ Last(xn)
        xcop ≜ Last(xe).cop
    IN  ∧ css′ = [css EXCEPT ![c].node = @ ∪ Range(xn),
                             ![c].edge = @ ∪ Range(xe)]
        ∧ ccur′ = [ccur EXCEPT ![c] = xcur]
        ∧ state′ = [state EXCEPT ![c] = 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]
        op with the sctx field (the extended part)
    ∧ LET cop ≜ [op ↦ op, oid ↦ [c ↦ c, seq ↦ cseq′[c]], ctx ↦ ccur[c], sctx ↦ {}]
      IN  ∧ ClientPerform(cop, c, Remote)
          ∧ comm!CSend(cop)

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

DoDel(c) ≜
    ∃ del ∈ Del :
        ∧ del.pos ∈ 1 .. Len(state[c])
        ∧ DoOp(c, del)
        ∧ UNCHANGED ⟨sssVars, eVars⟩

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

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

```
Rev(c) ≜
    ∧ comm!CRev(c)
    ∧ LET cop ≜ Head(cincoming[c])  the received (transformed) operation
      IN  ClientPerform(cop, c, Local)
    ∧ UNCHANGED ⟨eVars, cVars, sssVars⟩
```

The *Server* performs operation cop.

```
ServerPerform(cop) ≜
  LET c ≜ cop.oid.c
      xss ≜ xForm(cop, sss[c], scur[c], Remote)
      xn ≜ xss.node
      xe ≜ xss.edge
      xcur ≜ Last(xn)
      xcop ≜ Last(xe).cop
  IN  ∧ sss' = [cl ∈ Client ↦
                  IF cl = c
                  THEN [sss[cl] EXCEPT !.node = @ ∪ Range(xn),
                                       !.edge = @ ∪ Range(xe)]
                  ELSE LET scurcl ≜ scur[cl]
                           scurclprime ≜ scurcl ∪ {cop.oid}
                       IN  [sss[cl] EXCEPT !.node = @ ∪ {scurclprime},
                                           !.edge = @ ∪ {[from ↦ scurcl, to ↦ scurclprime,
                                                          cop ↦ xcop, lr ↦ Remote]}]
                ]
      ∧ scur' = [cl ∈ Client ↦
                  IF cl = c THEN xcur ELSE scur[cl] ∪ {cop.oid}]
      ∧ state' = [state EXCEPT ![Server] = Apply(xcop.op, @)]
      ∧ comm!SSendSame(c, xcop)  broadcast the transformed operation
```

The *Server* receives a message.

```
SRev ≜
    ∧ comm!SRev
    ∧ LET cop ≜ [Head(sincoming) EXCEPT !.sctx = soids]
        cop with the sctx field (the extended part)
      IN  ServerPerform(cop)
    ∧ UNCHANGED ⟨eVars, cVars, cssVars⟩
```

The next-state relation.

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

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

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

In *Jupiter* (not limited to *XJupiter*), each client synchronizes with the server. In *XJupiter*, this is expressed as the following *CSSync* property.

```
293 ASSUME (TLCSet(1, ⟨"SameOids", 0⟩))
294 CSSync ≜
295     ∀ c ∈ Client :
296         (ccur[c] = scur[c] ∧ TLCCnt(1, 100)) ⇒ css[c] = sss[c]
297 ╚═══════════════════════════════════════════════════════════════
```

* Modification History
* *Last* modified *Tue Oct* 30 20:46:48 *CST* 2018 by *hengxin*
* Created *Tue Oct* 30 20:32:27 *CST* 2018 by *hengxin*