1 MODULE *CJupiter*

Model of our own *CJupiter* protocol.

6 EXTENDS *Integers*, *OT*, *TLC*, *AdditionalFunctionOperators*

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 $Replica \triangleq Client \cup \{Server\}$

16 $List \triangleq Seq(Char \cup Range(InitState))$ all possible lists/strings

17 $MaxLen \triangleq Cardinality(Char) + Len(InitState)$ the max length of lists in any states;

18 We assume that all inserted elements are unique.

20 $ClientNum \triangleq Cardinality(Client)$

21 $Priority \triangleq \text{CHOOSE } f \in [Client \to 1 \,..\, ClientNum] : Injective(f)$

22

23 ASSUME

24 $\land\ Range(InitState) \cap Char = \{\}$

25 $\land\ Priority \in [Client \to 1 \,..\, ClientNum]$

26

The set of all operations. Note: The positions are indexed from 1.

31 $Rd \triangleq [type : \{\text{“Rd”}\}]$

32 $Del \triangleq [type : \{\text{“Del”}\}, pos : 1 \,..\, MaxLen]$

33 $Ins \triangleq [type : \{\text{“Ins”}\}, pos : 1 \,..\, (MaxLen + 1), ch : Char, pr : 1 \,..\, ClientNum]$ *pr*: priority

35 $Op \triangleq Ins \cup Del$

36

41 $Oid \triangleq [c : Client, seq : Nat]$ operation identifier

42 $Cop \triangleq [op : Op \cup \{Nop\}, oid : Oid, ctx : \text{SUBSET } Oid, sctx : \text{SUBSET } Oid]$ operation with context

44 $cop1 \prec cop2 \triangleq$

45 $\lor\ cop2.sctx = \{\}$

46 $\lor\ cop1.oid \in cop2.sctx$

48 $COT(lcop, rcop) \triangleq$

49 $[op \mapsto Xform(lcop.op, rcop.op), oid \mapsto lcop.oid,$

50 $ctx \mapsto lcop.ctx \cup \{rcop.oid\}, sctx \mapsto lcop.sctx]$

51

52 VARIABLES

For the client replicas:

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

57 *cstate*, *cstate*[*c*]: state (the list content) of the client $c \in Client$

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

86 $vars \triangleq \langle eVars, cVars, sVars, commVars, css, cur \rangle$ all variables

87 ----

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.

98 $IsCSS(G) \triangleq$

99 $\land\ G = [node \mapsto G.node,\ edge \mapsto G.edge]$

100 $\land\ G.node \subseteq (\text{SUBSET}\ Oid)$

101 $\land\ G.edge \subseteq [from : G.node,\ to : G.node,\ cop : Cop]$

103 $TypeOK \triangleq$

For the client replicas:

107 $\land\ cseq \in [Client \to Nat]$

108 $\land\ cstate \in [Client \to List]$

For the server replica:

112 $\land\ soids \subseteq Oid$

113 $\land\ sstate \in List$

For all replicas: the n-ary ordered state space

117 $\land \forall\, r \in Replica : IsCSS(css[r])$

118 $\land\ cur \in [Replica \to \text{SUBSET}\ Oid]$

For communication between the server and the clients:

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

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

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

```
180         ∧ UNCHANGED ⟨sVars, eVars⟩

182 Do(c) ≜
183       ∨ DoIns(c)
184       ∨ DoDel(c)

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

    rcss: the css at replica r ∈ Replica

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

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

195 RECURSIVE xFormHelper(_, _, _, _, _, _)
196 xFormHelper(u, v, cop, r, ns, es) ≜
197     IF u = cur[r]
198      THEN ⟨ns ∪ {v}, es ∪ {[from ↦ u, to ↦ v, cop ↦ cop]}, cop, v⟩
199      ELSE LET fedge ≜ CHOOSE e ∈ css[r].edge :
200                                 ∧ e.from = u
201                                 ∧ ∀ ue ∈ css[r].edge :
202                                     (ue.from = u ∧ ue ≠ e) ⇒ (e.cop ≺ ue.cop)
203                uprime ≜ fedge.to
204                fcop   ≜ fedge.cop
205                cop2fcop ≜ COT(cop, fcop)
206                fcop2cop ≜ COT(fcop, cop)
207                 vprime ≜ v ∪ {fcop.oid}
208           IN    xFormHelper(uprime, vprime, cop2fcop, r,
209                    ns ∪ {v},
210                    es ∪ {[from ↦ u, to ↦ v, cop ↦ cop],
211                          [from ↦ v, to ↦ vprime, cop ↦ fcop2cop]})

213 xForm(cop, r) ≜
214   LET
215     u ≜ Locate(cop, css[r])
216     v ≜ u ∪ {cop.oid}
217   IN
218    css′ = [css EXCEPT ![r].node = @ ∪ {v},
219                       ![r].edge = @ ∪ {[from ↦ u, to ↦ v, cop ↦ cop]}]

    Client c ∈ Client receives a message from the Server.

224 Rev(c) ≜
225       ∧ comm!CRev(c)
226       ∧ LET cop ≜ Head(cincoming[c])  the received original operation
227              u ≜ Locate(cop, css[c])
228              v ≜ u ∪ {cop.oid}
229           xcss ≜ xFormHelper(u, v, cop, c, {}, {})      the transformed operation
230         IN    ∧ css′ = [css EXCEPT ![c].node = @ ∪ xcss[1],
231                                    ![c].edge = @ ∪ xcss[2]]
```

```
                ∧ cur′ = [cur EXCEPT ![c] = xcss[4]]
                ∧ cstate′ = [cstate EXCEPT ![c] = Apply(xcss[3].op, @)]
        ∧ UNCHANGED ⟨cseq, sVars, eVars⟩
```

The *Server* receives a message.

```
SRev ≜
    ∧ comm!SRev
    ∧ LET org ≜ Head(sincoming)   the received operation
          cop ≜ [org EXCEPT !.sctx = soids]   set its sctx field
            u ≜ Locate(cop, css[Server])
            v ≜ u ∪ {cop.oid}
         xcss ≜ xFormHelper(u, v, cop, Server, {}, {})
      IN ∧ soids′ = soids ∪ {cop.oid}
         ∧ css′ = [css EXCEPT ![Server].node = @ ∪ xcss[1],
                              ![Server].edge = @ ∪ xcss[2]]
         ∧ cur′ = [cur EXCEPT ![Server] = xcss[4]]
         ∧ sstate′ = Apply(xcss[3].op, sstate)   apply the transformed operation
         ∧ comm!SSendSame(cop.oid.c, cop)   broadcast the original operation
    ∧ UNCHANGED ecVars
```

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)
```

```
\* Modification History
\* Last modified Mon Sep 03 19:22:40 CST 2018 by hengxin
\* Created Sat Sep 01 11:08:00 CST 2018 by hengxin
```