MODULE *CJupiter*

Specification of our own *CJupiter* protocol.

$\textsc{extends}\ Integers,\ OT,\ TLC,\ AdditionalFunctionOperators,\ AdditionalSequenceOperators$

$\textsc{constants}$

$\quad Client,$ the set of client replicas
$\quad Server,$ the (unique) server replica
$\quad Char,$ set of characters allowed
$\quad InitState$ the initial state of each replica

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

$List \triangleq Seq(Char \cup Range(InitState))$ all possible lists/strings
$MaxLen \triangleq Cardinality(Char) + Len(InitState)$ the max length of lists in any states;
We assume that all inserted elements are unique.

$ClientNum \triangleq Cardinality(Client)$
$Priority \triangleq \textsc{choose}\ f \in [Client \rightarrow 1 \mathrel{..} ClientNum] : Injective(f)$

$\textsc{assume}$
$\quad \land Range(InitState) \cap Char = \{\}$ due to the uniqueness requirement
$\quad \land Priority \in [Client \rightarrow 1 \mathrel{..} ClientNum]$

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

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

$Op \triangleq Ins \cup Del$

Cop: operation of type *Op* with context

$Oid \triangleq [c : Client,\ seq : Nat]$ operation identifier
$Cop \triangleq [op : Op \cup \{Nop\},\ oid : Oid,\ ctx : \textsc{subset}\ Oid,\ sctx : \textsc{subset}\ Oid]$

tb: Is $cop1$ totally ordered before $cop2$?

At a given replica $r \in Replica$, these can be determined in terms of $sctx$.

$tb(cop1,\ cop2,\ r) \triangleq$
$\quad \lor\ cop1.oid \in cop2.sctx$
$\quad \lor\ \land cop1.oid \notin cop2.sctx$
$\quad\quad\ \ \land cop2.oid \notin cop1.sctx$
$\quad\quad\ \ \land cop1.oid.c \neq r$

OT of two operations of type Cop.

$COT(lcop,\ rcop) \triangleq [lcop\ \textsc{except}\ !.op = Xform(lcop.op,\ rcop.op),\ !.ctx = @ \cup \{rcop.oid\}]$

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:

$incoming,$ $incoming[r]$: incoming channel of replica $r \in Replica$

For model checking:

$chins$ a set of chars to insert

$comm \triangleq \text{INSTANCE } CSComm \text{ WITH } Msg \leftarrow Cop,\ incoming \leftarrow incoming$

$eVars \triangleq \langle chins \rangle$ 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

$commVars \triangleq \langle incoming \rangle$ 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.

$IsCSS(G) \triangleq$

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

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

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

$TypeOK \triangleq$

For the client replicas:

$\quad \land\ cseq \in [Client \rightarrow Nat]$

For the server replica:

$\quad \land\ soids \subseteq Oid$

For all replicas: the n-ary ordered state space

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

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

$\quad \land\ state \in [Replica \rightarrow List]$

```
    For communication between the server and the clients:
    ∧ comm!TypeOK
    For model checking:
    ∧ chins ⊆ Char
├──────────────────────────────────────────────────────────────┤
The Init predicate.
Init ≜
    ∧ chins = Char
    For the client replicas:
    ∧ cseq = [c ∈ Client ↦ 0]
    For the server replica:
    ∧ soids = {}
    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!EmptyChannel
├──────────────────────────────────────────────────────────────┤
Locate the node in rcss which matches the context ctx of cop.

rcss: the css at replica r ∈ Replica
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.
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) ≜
            IF uh = cur[r]
             THEN xcss
             ELSE LET fedge ≜ CHOOSE e ∈ rcss.edge :
                                    ∧ e.from = uh
                                    ∧ ∀ uhe  ∈ rcss.edge :
                                        (uhe.from = uh ∧ uhe ≠ e) ⇒ tb(e.cop, uhe.cop, 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⟩,
                                             the order of recording edges here is important
                                            !.edge = @ ∘ ⟨[from ↦ vh, to ↦ vprime, cop ↦ fcop2coph],
                                                    [from ↦ uprime, to ↦ vprime, cop ↦ coph2fcop]⟩])
    IN  xFormHelper(u, v, cop, [node ↦ ⟨v⟩,
                                edge ↦ ⟨[from ↦ u, to ↦ v, cop ↦ cop]⟩])
```

Perform cop at replica $r \in Replica$.

```
Perform(cop, r) ≜
    LET xcss ≜ xForm(cop, r)
        xn ≜ xcss.node
        xe ≜ xcss.edge
        xcur ≜ Last(xn)
        xcop ≜ Last(xe).cop
    IN  ∧ css′ = [css EXCEPT ![r].node = @ ∪ Range(xn),
                             ![r].edge = @ ∪ Range(xe)]
        ∧ cur′ = [cur EXCEPT ![r] = xcur]
        ∧ state′ = [state EXCEPT ![r] = 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]
      ∧ LET cop ≜ [op ↦ op, oid ↦ [c ↦ c, seq ↦ cseq′[c]], ctx ↦ cur[c], sctx ↦ {}]
          IN  ∧ Perform(cop, c)
              ∧ 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 sVars

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

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

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

```
Rev(c) ≜
    ∧ comm!Rev(c)
    ∧ LET cop ≜ Head(incoming[c])    the received original operation
       IN   Perform(cop, c)
    ∧ UNCHANGED ⟨ecVars, sVars⟩
------------------------------------------------------------------------
The Server receives a message.
SRev ≜
    ∧ PrintT(comm!Rev(Server))
    ∧ comm!Rev(Server)
    ∧ LET cop ≜ [Head(incoming[Server]) EXCEPT !.sctx = soids]    set its sctx field
       IN   ∧ soids′ = soids ∪ {cop.oid}
            ∧ Perform(cop, Server)
            ∧ 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)
------------------------------------------------------------------------
The compactness of CJupiter: the css at all replicas are essentially the same.
IgnoreSctx(rcss) ≜
    [rcss EXCEPT !.edge = {[e EXCEPT !.cop.sctx = {}] : e ∈ @}]

Compactness ≜
    comm!EmptyChannel ⇒ Cardinality({IgnoreSctx(css[r]) : r ∈ Replica}) = 1

THEOREM Spec ⇒ Compactness
========================================================================
\* Modification History
\* Last modified Sat Sep 08 15:55:43 CST 2018 by hengxin
\* Created Sat Sep 01 11:08:00 CST 2018 by hengxin
```