Wei HF, Tang RZ, Huang Y *et al.* Jupiter made abstract, and then refined. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(6): 1343–1364 Nov. 2020. DOI 10.1007/s11390-020-0516-0

Jupiter Made Abstract, and Then Refined

Heng-Feng Wei, *Member, CCF*, Rui-Ze Tang, Yu Huang*, *Member, CCF*, and
Jian Lv, *Fellow, CCF, Member, ACM*

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China

E-mail: hfwei@nju.edu.cn; tangruize97@gmail.com; {yuhuang, lj}@nju.edu.cn

Received April 10, 2020; revised October 22, 2020.

Abstract Collaborative text editing systems allow multiple users to concurrently edit the same document, which can be modeled by a replicated list object. In the literature, there is a family of operational transformation (OT)-based Jupiter protocols for replicated lists, including AJupiter, XJupiter, and CJupiter. They are hard to understand due to the subtle OT technique, and little work has been done on formal verification of complete Jupiter protocols. Worse still, they use quite different data structures. It is unclear about how they are related to each other, and it would be laborious to verify each Jupiter protocol separately. In this work, we make contributions towards a better understanding of Jupiter protocols and the relation among them. We first identify the key OT issue in Jupiter and present a generic solution. We summarize several techniques for carrying out the solution, including the data structures to maintain OT results and to guide OTs. Then, we propose an implementation-independent AbsJupiter protocol. Finally, we establish the (data) refinement relation among these Jupiter protocols (AbsJupiter included). We also formally specify and verify the family of Jupiter protocols and the refinement relation among them using TLA$^+$ (TLA stands for "Temporal Logic of Actions") and the TLC model checker. To our knowledge, this is the first work to formally specify and verify a family of OT-based Jupiter protocols and the refinement relation among them. It would be helpful to promote a rigorous study of OT-based protocols.

Keywords Jupiter protocol, operational transformation, refinement, replicated list, TLA$^+$

1 Introduction

Collaborative text editing systems, such as Google Docs①, Firepad②, Overleaf③, and SubEthaEdit④, allow multiple users to concurrently edit the same document. For availability, such systems often replicate the document at several replicas. For low latency, replicas are required to respond to user operations immediately and updates are propagated asynchronously [1, 2].

The replicated list object is frequently used to model the core functionality (e.g., insertion and deletion) of replicated collaborative text editing systems [1–4]. A common specification for it is strong eventual consistency (SEC) [3]. It requires that whenever two replicas have processed the same set of updates, they have the same list. A family of Jupiter protocols [3] for implementing such a replicated list have been proposed, including XJupiter [4] (a multi-client version of [3] given by Xu *et al.*), AJupiter [2] (another multi-client version of [3] given by Attiya *et al.*), and CJupiter [6] (short for Compact Jupiter). They adopt the client/server (C/S) architecture, where the server serializes operations and propagates them from one client to others (Fig.1). Note that since replicas are required to respond to user operations immediately, the C/S architecture does not im-

Regular Paper

Special Section on Software Systems 2020

This work was (partially) supported by the National Natural Science Foundation of China under Grant Nos. 61690204, 61932021, 61702253, and 61772258.

*Corresponding Author

① GoogleDocs. https://docs.google.com, Sept. 2020.

② Firepad. https://firepad.io/, Sept. 2020.

③ Overleaf. https://www.overleaf.com/, Sept. 2020.

④ SubEthaEdit. https://subethaedit.net/, Sept. 2020.

ply that clients process operations in the same order. To achieve convergence, Jupiter adopts the operational transformation (OT) technique[1,7] to resolve the conflicts caused by concurrent operations. The idea of OT is, for each replica, to process local operations immediately and to transform received operations according to the effects of previously processed concurrent operations. The transformation rules are called OT functions[1,3].

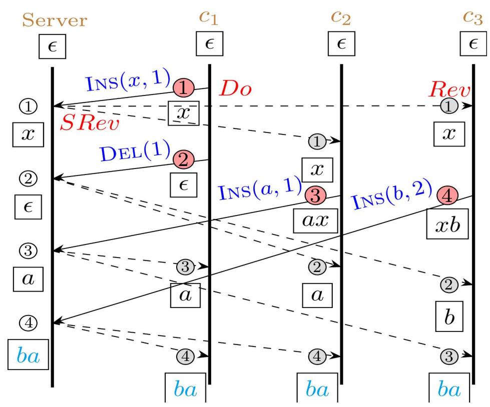


Fig.1. System model. The circled numbers indicate the serialization order (so) in which the operations are received at the server (Section 3). The list produced by Jupiter protocols are shown in boxes[6].

Example 1 (*Illustration of OT*). Fig.2 shows a replicated list system with two client replicas C_1 and C_2 which initially hold the same list "ab". Suppose that user 1 issues $o_1 = \text{Ins}(1, x)$ at C_1 and concurrently user 2 issues $o_2 = \text{Del}(2)$ at C_2. After being executed locally, each operation is sent to the other replica. Without OT, C_1 and C_2 wind up with different lists (i.e., "xb" and "xa", respectively). With OT, o_2 is transformed to $o'_2 = \text{Del}(3)$ at C_1, taking into account the fact that o_1 has inserted an element at position 1. Meanwhile, o_1 remains unchanged after OT at C_2. As a result, two replicas converge to the same list "xa".

When several replicas diverge by multiple operations, OT becomes much more subtle and error-prone. Some published OT-based protocols[1,8] were even later shown incorrect[9–11]. The intrinsic complexity in concurrency control makes the OT-based Jupiter protocols hard to understand. Moreover, little has been done on the formal verification of complete OT-based protocols (not only of OT functions). Worse still, Jupiter protocols use quite different data structures, rendering the relation among them unclear. It would be also laborious and wasteful to prove or verify that the Jupiter protocols satisfy a certain property one by one. In this work, we make the following contributions towards a better understanding of Jupiter protocols and the relation among them (Fig.3).

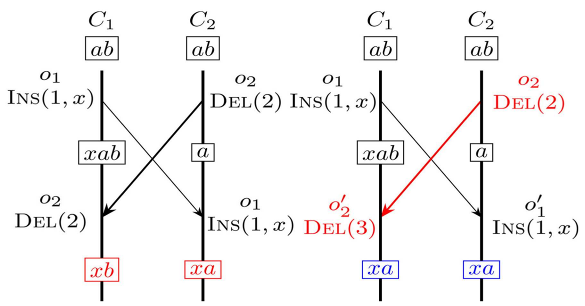


Fig.2. Example for OT. The positions are indexed from 1. (The server is not shown.) (a) Without OT, the states of C_1 and C_2 diverge. (b) With OT, C_1 and C_2 converge to the same state.

• We first identify the key issue involving OT that Jupiter needs to address as follows: when a replica r receives an operation op, which operations should op be transformed against and in what order before it is applied? We also present a generic solution to this issue: transform op against the set of concurrent operations previously executed at r in the serialization order established at the server. Then, we summarize several techniques that the Jupiter protocols adopt to carry out the solution, including those for deciding whether two operations are concurrent, those for determining the serialization order, and the data structures to maintain (intermediate) OT results and to guide OTs.

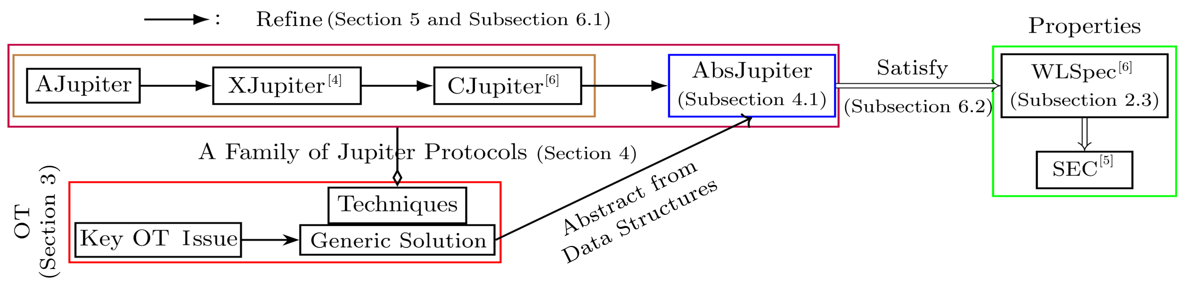


Fig.3. Overview of contributions.

• We propose AbsJupiter, an abstract Jupiter protocol which captures the OT essence of existing Jupiter protocols. It addresses the key OT issue in a way which is abstract from concrete data structures by using mathematical sets.

• For different purposes such as performance or ease of correctness proof, existing Jupiter protocols use quite different data structures. The implementation details in data structures have obscured the similarities among them. We show that the existing Jupiter protocols are actually (data) refinements [12–14] of AbsJupiter in data structures. Specifically, we show that AJupiter is a refinement (a.k.a. implementation) of XJupiter, XJupiter is a refinement of CJupiter, and CJupiter is a refinement of AbsJupiter. As a consequence, the properties like SEC and WLSpec (weak list specification defined in Subsection 2.3) that hold for AbsJupiter also automatically hold for other Jupiter protocols.

• We formally specify the family of Jupiter protocols and the refinement mappings among them in TLA$^+$ [15]⑤. Finally, we present the model checking results conducted by TLC [16] (the model checker [17] for TLA$^+$) of verifying both the properties for Jupiter protocols and refinement relations among them.

Section 2 provides a brief introduction to TLA$^+$ and covers preliminaries on system model, OT, and list specifications. Section 3 identifies the key OT issue in Jupiter and presents a generic solution. Section 4 describes the family of Jupiter protocols, including AbsJupiter. Section 5 establishes the refinement relation among Jupiter protocols. Section 6 presents the model checking results. Section 7 discusses related work. Section 8 concludes the paper.

2 Preliminaries

2.1 TLA$^+$

The specification language TLA$^+$ was designed by Lamport for modelling and reasoning about concurrent and distributed programs [15]. In TLA$^+$, systems are modelled as state machines. A state machine is described by its initial states and actions. A state is an assignment of values to variables. An action is a relation between old states and new states, and is represented by a formula over unprimed variables referring to the old state and primed variables referring to the new state. For example, $x' = y + 42$ is the relation asserting that the value of x in the new state is 42 greater than that of y in the old state.

TLA$^+$ is based on TLA, the Temporal Logic of Actions [18]. A program is specified in TLA$^+$ as a temporal formula of TLA of the form $Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge L$, where $Init$ is a predicate specifying all possible initial states of the program, $Next$ specifies the next-state relation of the program, $\Box$ is the temporal operator read "Always", $vars$ is the tuple of all variables used in the program, and L is a fairness property (not used in this paper). The next-state relation $Next$ is typically a disjunction of all the actions of the program. The expression $[Next]_{vars}$ is true if $Next$ is true, meaning that some action is true and thus taken, or if $vars$ stutters, meaning that their values are unchanged. A behavior of the program specified by $Spec$ (ignoring L) of the above form is a sequence of states that satisfy $Spec$, namely, the $Init$ predicate holds in the first state of this sequence, and the next-state relation $[Next]_{var}$ holds for any two consecutive states of this sequence.

TLA$^+$ combines TLA with the first-order logic and Zermelo-Fraenkel set theory. Table 1 summarizes the operators in the logic and set theory we use in this paper. It is an excerpt from the complete summary of TLA$^+$⑥ and shows only the operators that have special notations in TLA$^+$.

Specifications of programs are grouped into modules. In a module, we can declare constants (CONSTANTS) and variables (VARIABLES), define operators ($F(x_1, \cdots, x_n) \triangleq \cdots$), and claim theorems (THEOREM). A module M can import the declarations, definitions, and theorems from other modules $M_1, \cdots, M_n$ by extending them, namely writing EXTENDS $M_1, \cdots, M_n$ in M. Modules can also be instantiated. Let us consider the following INSTANCE statement in module M:

$$IM_1 \triangleq \text{INSTANCE } M_1 \text{ WITH } p_1 \leftarrow e_1, \cdots, p_n \leftarrow e_n,$$

where p_i consists of all declared constants and variables of M_1 and e_i are valid expressions in M⑦. For each operator F and its definition d of module M_1, this defines F to be the operator, denoted by $IM_1!F$, whose

⑤https://github.com/hengxin/jupiter-refinement-project, Sept. 2020.

⑥Leslie Lamport. Summary of TLA$^+$. http://lamport.azurewebsites.net/tla/summary-standalone.pdf, Sept. 2020.

⑦Note that constant parameters p_i must be instantiated by constant-level expressions built up from constants and constant operators and variable parameters by state-level expressions which may contain variables and the ENABLED operator (not used in this paper). For simplicity, we omit the formal definitions of levels [15].

Table 1. Summary of TLA$^+$ Operators Used in This Paper

Category	Operator	Meaning
Logic	CHOOSE $x \in S : P(x)$	x in S satisfying $P(x)$⑧
Set	SUBSET S	Powerset (i.e., set of subsets) of S
	$\{e : x \in S\}$	Set of elements e such that x is in S
	$\{x \in S : p\}$	Set of elements x in S satisfying p
Function	$f[e]$	Function application
	$[x \in S \mapsto e]$	Function f such that $f[x] = e$ for $x \in S$
	$[f$ EXCEPT $![e_1] = e_2]$, where e_2 may contain @	Function $\hat{f}$ equals f except that $\hat{f}[e_1] = e_2$, where any occurrence of @ in e_2 stands for $f[e_1]$
Record	$e.h$	The h-field of record e
	$[h_1 \mapsto e_1, \cdots, h_n \mapsto e_n]$	The record whose h_i field is e_i
	$[h_1 : S_1, \cdots, h_n : S_n]$	Set of all records with h_i field in S_i
	$[r$ EXCEPT $!.h = e]$, where e may contain @	Record $\hat{r}$ equals r except that $\hat{r}.h = e$, where any occurrence of @ in e stands for $r.h$
Tuple	$e[i]$	The i-th component of tuple e
	$\langle e_1, \cdots, e_n \rangle$	The n-tuple whose i-th component is e_i
Sequence	$Head(s)$	The first element of sequence s
	$Last(s)$	The last element of sequence s
	$Tail(s)$	The tail of sequence s, which consists of s with its head removed
	$Range(s)$	The set of elements of sequence s
Action operator	e'	The value of e in the new state of an action
	$[A]_e$	$A \vee (e' = e)$
Temporal operator	$\Box F$	F is always true

definition is obtained from d by replacing each p_i with e_i.

TLC is an explicit-state model checker for TLA$^+$ [16]. It can compute and explore the state space of finite-state instances of TLA$^+$ specifications. These finite-state instances are called TLC models of TLA$^+$ specifications. For example, a TLC model of a specification describing a distributed system consisting of a set of processors declared as CONSTANTS $Proc$ should instantiate $Proc$ with a set consisting of a fixed number of processors, like $Proc \triangleq \{1, 2, 3\}$. We can also represent a process by a TLC model value, which is considered to be unequal to any other values in TLA$^+$. Therefore, we can instantiate $Proc$ with a set of model values $Proc \triangleq \{p1, p2, p3\}$. Moreover, if permuting the elements in a set of model values does not change whether a behavior satisfies a desired specification, we can further use the symmetry set technique to reduce the state space that TLC has to check [15].

In TLA$^+$, refinement is logical implication. Suppose we have two specifications: $AbsSpec$ defined in module $AbsModule$ with variables $x_1, \cdots, x_m, y_1, \cdots, y_n$, and $ImplSpec$ defined in module $ImplModule$ with variables $x_1, \cdots, x_m, z_1, \cdots, z_p$. Let X, Y, and Z denote $x_1, \cdots, x_m$, $y_1, \cdots, y_n$, and $z_1, \cdots, z_p$, respectively. To verify that $ImplSpec$ refines $AbsSpec$, formally $ImplSpec \implies AbsSpec$, we need to show that for each behavior satisfying $ImplSpec$, there is some way to assign values of the variables Y in each state so that the resulting behavior satisfies $AbsSpec$ [13]. This can be done by explicitly specifying those values of Y in terms of X and Z. Specifically, for each y_i, we define an expression $\overline{y_i}$ in terms of X and Z, substitute $y_i \leftarrow \overline{y_i}$ in $AbsSpec$ to get $\overline{AbsSpec}$, and we show that $ImplSpec$ refines $\overline{AbsSpec}$. The substitution $y_i \leftarrow \overline{y_i}$ is called a refinement mapping. To verify the assertion that $ImplSpec$ refines $AbsSpec$ under such a refinement mapping in TLA$^+$, we can add the following definition to module $ImplModule$ ($AbsSub$ is a fresh identifier).

$$AbsSub \triangleq \text{INSTANCE } AbsModule \text{ WITH } y_1 \leftarrow \overline{y_1}, \cdots, y_n \leftarrow \overline{y_n}.$$

Then we let TLC check the theorem:

$$\text{THEOREM } ImplSpec \implies AbsSub!AbsSpec,$$

which is added to module $ImplModule$.

There are two kinds of refinement [14], namely data refinement [12] and step refinement. In data refinement, the "abstract" data of a high-level protocol is refined by a "concrete" representation of a lower-level protocol [12]. In step refinement, a single step (i.e., actions in terms of TLA$^+$) of a high-level protocol is refined by multiple steps of a lower-level protocol [14].

⑧ The most common use of the CHOOSE operator is to select a unique value satisfying $P(x)$ [15]. If there is no element $x \in S$ satisfying $P(x)$, then TLC will report an error. On the other hand, if there are several such x's, then an arbitrary one is chosen.

Constructing a refinement mapping may require adding auxiliary variables to the (lower-level) protocols [13, 19]. One kind of auxiliary variables that we will use in data refinement among Jupiter protocols is called history variables [13, 19]. Intuitively, history variables record the information about past behaviors of a protocol, and are typically not used by the actual variables of the protocol. Therefore, it is safe to add history variables to protocols, without altering their behaviors [13].

2.2 System Model

We let $Client$ denote the set of client replicas, $Server$ the unique server replica, and $Replica \triangleq Client \cup \{Server\}$ the set of all replicas. Client replicas are connected to the server replica via FIFO channels. The set of messages is denoted by M. A replica is modelled as a state machine. Each replica r maintains its current list $list[r]$ (initially empty; denoted by ϵ) and interacts with three kinds of actions from users and other replicas.

- $Do(c \in Client, op \in Op)$. Client c receives an operation $op \in Op$ (defined in Subsection 2.3) from an unspecified user (we also sometimes say that client c generates the operation op) and responds to the user immediately. It then sends the update in a message $m \in M$ to the server asynchronously.
- $Rev(c \in Client, m \in M)$. Client c receives and processes a message m from the server.
- $SRev(m \in M)$. The server receives a message m from a client. It will produce and broadcast a new message to other clients.

Example 2 (*Behaviors of Replicas*). We consider client c_3 in Fig.1. First, in $Rev(c_3, \_)$, client c_3 receives a message containing the information about o_1 (maybe transformed) of client c_1 from the server. Next, in $Do(c_3, o_4)$, it generates operation o_4 ($\textsc{Ins}(b, 2)$), applies o_4 locally, and sends o_4 to the server. Then, in $Rev(c_3, \_)$, it receives messages containing the information about o_2 and o_3 of clients c_1 and c_2 respectively, from the server. The list $list[c_3]$ at c_3 is updated accordingly.

2.3 List, OT, and Weak List Specification

A replicated list object supports two types of update operations: Del and Ins, defined as records in module Op (Fig.4). Following [2], we assume that all inserted elements are unique, which can be achieved by attaching replica identifiers and local sequence numbers. The priority field "pr" of Ins helps to resolve the conflicts caused by two concurrent Ins operations that are intended to insert different elements at the same position.

Module OT (Fig.5) shows a complete definition of OT functions for lists [1, 3]. $OT(lop, rop)$ transforms lop against rop by calling the appropriate OT function according to the types of lop and rop. For example, $OTID$ defines how an Ins operation ins is transformed against a Del operation del. It adjusts the insertion position of ins according to the deletion position of del.

We consider the weak list specification WLSpec [2], which is stronger than strong eventual consistency (SEC) [5]. WLSpec is equivalent to the "pairwise state compatibility property" [6]. It requires any pair of lists across the system to be compatible. Two lists l_1 and l_2 are compatible if for any two common elements e_1 and e_2 of l_1 and l_2, the relative ordering of e_1 and e_2 is the same in l_1 and l_2 (see module $WLSpec$ (Fig.6) for the formal specification of $Compatible$). Let $hlist$ be a set of lists. WLSpec is defined as $WLSpec \triangleq \forall l_1, l_2 \in hlist : Compatible(l_1, l_2)$ (see also module AbsJupiterH in Subsection 6.2).

Example 3 (*Weak List Specification. Adapted from* [6]). We consider the execution in Fig.1. There exist three replica states with lists $l_1 = ba$, $l_2 = ax$, and $l_3 = xb$, respectively. This is allowed by WLSpec, since the lists are pairwise compatible. However, an execution is not allowed by WLSpec, if it contained two states with, say, $l = ab$ and $l' = ba$.

3 Jupiter Family

The key issue for Jupiter protocols to address is as follows. When a replica r receives an operation op, which operations should op be transformed against and in what order before it is applied? The solution is to

MODULE Op

$Del \triangleq [type : \{\text{"Del"}\}, pos : Nat]$ The positions (pos) are indexed from 1.
$Ins \triangleq [type : \{\text{"Ins"}\}, pos : Nat, ch : Char, pr : 1 \mathrel{..} Cardinality(Client)]$
$Op \triangleq Ins \cup Del$ The set of all possible update operations.
$Nop \triangleq \text{CHOOSE } o : o \notin Op$

Fig.4. TLA+ module Op.

```
──────────────────── MODULE OT ────────────────────
OTII(lins, rins) ≜   lins is transformed against rins; II is for Ins vs. Ins.
    IF lins.pos < rins.pos THEN lins
     ELSE IF lins.pos > rins.pos
            THEN [lins EXCEPT !.pos = @ + 1]
            ELSE IF lins.ch = rins.ch THEN Nop
                   ELSE IF lins.pr > rins.pr THEN lins  using "priority"
                          ELSE [lins EXCEPT !.pos = @ + 1]
OTID(ins, del) ≜   ins is transformed against del
    IF ins.pos ≤ del.pos THEN ins
                          ELSE [ins EXCEPT !.pos = @ − 1]
OTDI(del, ins) ≜   del is transformed against ins
    IF del.pos < ins.pos THEN del
                          ELSE [del EXCEPT !.pos = @ + 1]
OTDD(ldel, rdel) ≜   ldel is transformed against rdel; DD is for Del vs. Del.
    IF ldel.pos < rdel.pos THEN ldel
     ELSE IF ldel.pos = rdel.pos THEN Nop
            ELSE [ldel EXCEPT !.pos = @ − 1]
OT(lop, rop) ≜   lop is transformed against rop
    CASE lop = Nop ∨ rop = Nop → lop
      □ lop.type = "Ins" ∧ rop.type = "Ins" → OTII(lop, rop)
      □ lop.type = "Ins" ∧ rop.type = "Del" → OTID(lop, rop)
      □ lop.type = "Del" ∧ rop.type = "Ins" → OTDI(lop, rop)
      □ lop.type = "Del" ∧ rop.type = "Del" → OTDD(lop, rop)
───────────────────────────────────────────────────
```

Fig.5. TLA$^+$ module *OT*.

```
──────────────────── MODULE WLSpec ────────────────────
Compatible(l1, l2) ≜   Are l1 and l2 compatible?
    ∨ l1 = l2  Obviously true
    ∨ LET commonElements ≜ Range(l1) ∩ Range(l2)
       IN ∀ e1, e2 ∈ commonElements :
            ∨ e1 = e2
            ∨ FirstIndexOfElement(l1, e1) < FirstIndexOfElement(l1, e2)
              ≡ FirstIndexOfElement(l2, e1) < FirstIndexOfElement(l2, e2)
───────────────────────────────────────────────────────
```

Fig.6. TLA$^+$ module *WLSpec*.

transform *op* against the operations that are concurrent with it and have been previously executed at r in their serialization order, denoted by SO, i.e., the order in which they are received by the server. The four Jupiter protocols we study differ in the way they carry out the solution. Table 2 summarizes several key techniques that they adopt to carry out the solution, including those for deciding whether two operations are concurrent, those for determining the serialization order, and the data structures to maintain (intermediate) OT results and to guide OTs.

3.1 Context-Based OT (COT)

According to whether they use context-based operations (Cop) and context-based OT (COT) [20], Jupiter protocols fall into two categories: context-based including AbsJupiter, CJupiter, XJupiter, and non-context based, i.e., AJupiter. In this subsection, we define *Cop* and *COT*. How they are used to decide whether two operations are concurrent or not is explained in Subsection 3.3, along with the concrete data structures.

Table 2. Techniques Adopted by Jupiter Protocols to Address the Key OT Issue

Protocol	Concurrent Operation	SO Order	Data Structure
AbsJupiter	COT	SV	Set
CJupiter [6]	COT	SV	n-ary digraph
XJupiter [4]	COT	COT	2D digraph
AJupiter [2]	ACK	Buffer	1D buffer

Each operation $op \in Op$ is associated with a unique operation identifier (oid, for short) in *Oid*, which is a record of client c that generates *op* and a local sequence number $cseq[c]$ of c. Each replica r maintains their document state $ds[r]$ as the set of operation identifiers it has processed. The document state $ds[r]$ is updated to include *oid* whenever the replica r receives and processes an operation with *oid*.

Operations in $ds[r]$ of each replica r are related to each other via contexts. Intuitively, the context of an operation is a set of operations that it is aware of. Formally, in module *COT* (Fig.7), a context-based operation $cop \in Cop$ is a record of operation $op \in Op$, its oid $oid \in Oid$, and its context $ctx \subseteq Oid$ representing a document state. When an operation is generated by client c, its context is set to be the current document state $ds[c]$ of c. When a context-based operation $lcop$ is transformed against another one $rcop$, $lcop.ctx$ will be updated to include $rcop.oid$ (see module *COT*). Note that according to the context-based condition (CC)[20], two context-based operations can be transformed against each other, only if they have the same context. This will be guaranteed by context-based Jupiter protocols.

3.2 Serial Views (SV)

In AbsJupiter and CJupiter, replicas need to decide the SO order among operations (i.e., the order in which they are received by the server) with local knowledge. To do this, each replica r maintains a serial view $serial[r]$ which is a sequence of oids, representing its own knowledge about SO. The server always has the latest serial view $serial[Server]$ and updates it in $SRev$ by each time appending to it the recently received oid. In addition, $serial[Server]$ will be broadcast to clients along with actual messages. Each client c synchronizes its serial view with the server by updating $serial[c]$ to the latest $serial[Server]$ that it receives in $Rev(c, \_)$.

Let us consider two operation identifiers $oid1$ and $oid2$ that are generated or received by some replica r. The operator $so(oid1, oid2, sv)$ in module *SV* (Fig.8) decides whether $oid1$ precedes (or will precede) $oid2$ in SO order given the local serial view sv of r. There are three cases: 1) if both have been at the server, we use the order in which they arrived at the server, which is captured by the positions they are in sv; 2) if none has been at the server, they must be generated by the same client, and we use the order they were generated; 3) otherwise, the one that has been at the server precedes the other that has not.

3.3 Data Structures

3.3.1 Set

In AbsJupiter, each replica r maintains a set $copss[r]$ of context-based operations. When a replica r receives a context-based operation cop, it calls $xForm(r, cop)$ of module *Set* (Fig.9) to transform cop against a subset of context-based operations in $copss[r]$ that are concurrent with cop in their SO order.

Due to the FIFO communication, we have that $cop.ctx \subseteq ds[r]$. Thus, $xForm$ first calculates the set of (oids of) concurrent operations with cop as the set difference $ctxDiff$ between $ds[r]$ and $cop.ctx$. Then it recursively transforms cop against the context-based operations in $copss[r]$ whose oids are in $ctxDiff$ in their SO order according to the serial view $serial[r]$. This is done in $xFormHelper(coph, ctxDiffh, copssh)$.

1) If $ctxDiffh$ is empty, the most recently transformed $coph$ and the latest data structure $copssh$ are returned.

2) Otherwise, $xFormHelper$ chooses the next operation $fcoph$ against which $coph$ is to be transformed,

MODULE *COT*

$Oid \triangleq [c : Client,\ seq : Nat]$ the client that generates cop

$Cop \triangleq [op : Op,\ oid : Oid,\ ctx : \text{SUBSET } Oid]$ $ClientOf(cop) \triangleq cop.oid.c$

$COT(lcop,\ rcop) \triangleq$

$\quad [lcop \text{ EXCEPT } !.op = OT(lcop.op,\ rcop.op),\ !.ctx = @ \cup \{rcop.oid\}]$

Fig.7. TLA$^+$ module *COT*.

MODULE *SV*

$so(oid1,\ oid2,\ sv) \triangleq$ Is $oid1$ totally ordered before $oid2$ according to sv?

$\quad \text{LET } pos1 \triangleq FirstIndexOfElementSafe(sv,\ oid1)$ 0 if $oid1$ is not in sv

$\quad\quad\quad pos2 \triangleq FirstIndexOfElementSafe(sv,\ oid2)$ 0 if $oid2$ is not in sv

$\quad \text{IN}\quad \text{IF } pos1 \neq 0 \land pos2 \neq 0$ Case 1: both have been at the server

$\quad\quad\quad \text{THEN } pos1 < pos2$ using the order in which they arrived at the server

$\quad\quad\quad \text{ELSE IF } pos1 = 0 \land pos2 = 0$ Case 2: none has been at the server

$\quad\quad\quad\quad\quad \text{THEN } oid1.seq < oid2.seq$ using the order they were generated

$\quad\quad\quad\quad\quad \text{ELSE } pos1 \neq 0$ Case 3: the one that has been at the server is ahead

Fig.8. TLA$^+$ module *SV*.

```
MODULE Set
xForm(r, cop) ≜   Transform cop at replica r
  LET ctxDiff ≜ ds[r] \ cop.ctx   calculate concurrent operations
      xFormHelper(coph, ctxDiffh, copssh) ≜
        IF ctxDiffh = {} THEN [xcop ↦ coph, xcopss ↦ copssh]
         ELSE LET foidh ≜ CHOOSE oid ∈ ctxDiffh :
                            ∀ id ∈ ctxDiffh \ {oid} : so(oid, id, serial[r])
                  fcoph ≜ CHOOSE fcop ∈ copss[r] :
                            fcop.oid = foidh ∧ fcop.ctx = coph.ctx
                  xcoph ≜ COT(coph, fcoph)
                 xfcoph ≜ COT(fcoph, coph)
              IN   xFormHelper(xcoph, ctxDiffh \ {fcoph.oid},
                               copssh ∪ {xcoph, xfcoph})
  IN   xFormHelper(cop, ctxDiff, copss[r] ∪ {cop})
```

Fig.9. TLA$^+$ module *Set*.

such that $fcoph.oid$ is the first one in the current $ctxDiffh$ and $fcoph.ctx = coph.ctx$. Because the communication in the client/server model is FIFO, when an operation *cop* is received by some replica, the operations in its context have already been in this replica. Thus, such $fcoph$ satisfying $fcoph.ctx = cop.ctx$ exists. The existence of $fcoph$ in recursion can be further justified by induction.

3) *coph* and $fcoph$ are transformed against each other. The intermediate transformed operation *xcoph* is recursively transformed against the remaining concurrent operations (with *oid*) in $ctxDiffh \setminus \{foph.oid\}$.

3.3.2 Digraph

In CJupiter and XJupiter, the set of context-based operations is organized into edge-labeled digraphs. A digraph is represented by a record with *node* and *edge* fields (see $IsDigraph$ of module $Digraph$ (Fig.10)). Each node in $G.node$ of a digraph G represents a document state. Each directed edge e in $G.edge$ is labeled with a context-based operation *cop* satisfying $cop.ctx = e.from$, meaning that when applied, *cop* changes the document state from $e.from$ to $e.to = e.from \cup \{cop.oid\}$. The operator $\oplus$ takes the union of two records with *node* and *edge* fields.

```
MODULE Digraph
IsDigraph(G) ≜   G is a record with node and edge fields
    ∧ G.node ⊆ (SUBSET Oid)   each node represents a document state
    ∧ G.edge ⊆ [from : G.node, to : G.node, cop : Cop]
EmptyGraph ≜ [node ↦ {{}}, edge ↦ {}]

g ⊕ h ≜ [node ↦ g.node ∪ h.node, edge ↦ g.edge ∪ h.edge]

xForm(NextEdge(_, _, _), r, cop, g) ≜   Transform cop in g at replica r
  LET u ≜ CHOOSE n ∈ g.node : n = cop.ctx    v ≜ u ∪ {cop.oid}
      xFormHelper(uh, vh, coph, gh) ≜
        IF uh = ds[r] THEN [xcop ↦ coph, xg ↦ gh,
              lg ↦ [node ↦ {vh},
            edge ↦ {[from ↦ uh, to ↦ vh, cop ↦ coph]}]]
         ELSE LET e ≜ NextEdge(r, uh, g)   specific to CJupiter and XJupiter
                ecop ≜ e.cop     eu ≜ e.to     ev ≜ vh ∪ {ecop.oid}
           coph2ecop ≜ COT(coph, ecop)
           ecop2coph ≜ COT(ecop, coph)
              IN   xFormHelper(eu, ev, coph2ecop,
               gh ⊕ [node ↦ {ev},
                     edge ↦ {[from ↦ vh, to ↦ ev, cop ↦ ecop2coph],
                             [from ↦ eu, to ↦ ev, cop ↦ coph2ecop]}])
  IN   xFormHelper(u, v, cop, [node ↦ {v},
                      edge ↦ {[from ↦ u, to ↦ v, cop ↦ cop]}])
```

Fig.10. TLA$^+$ module *Digraph*.

In CJupiter and XJupiter, when a replica r (either client or server) receives a context-based operation *cop*, it calls $xForm(NextEdge, r, cop, g)$ of module *Digraph* to iteratively transform *cop* against a sequence of context-based operations along a path in some digraph g maintained by r. This path starts with the node u equal to *cop.ctx* and ends with the one equal to $ds[r]$. Each such path contains the operations whose oids are in $ds[r] \setminus cop.ctx$, which are concurrent with *cop* due to the FIFO communication. The next edge is chosen by $NextEdge$ specific to CJupiter and XJupiter to ensure the SO order. $xFormHelper(uh, vh, coph, gh)$ starts the transformation with $uh \leftarrow u$ (Fig.11 and module *Digraph*).

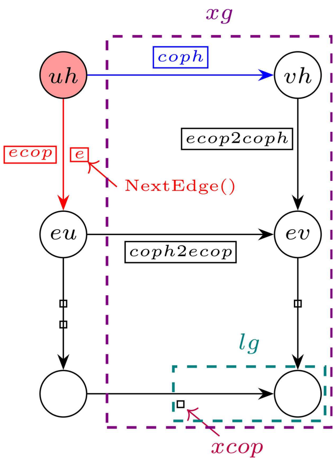


Fig.11. Illustration of $xForm$ of module *Digraph*.

1) If $uh = ds[r]$, the most recently transformed operation *coph*, the record *gh* consisting of nodes and edges produced in $xForm$ so far, and the node and the edge (collected in lg) produced in the last iteration of transformation are returned.

2) Otherwise, the next edge e outgoing from uh is chosen using $NextEdge(r, uh, g)$ specific to CJupiter and XJupiter.

3) *coph* and *ecop* are transformed against each other. The intermediate transformed operation *coph2ecop* is then recursively transformed against the sequence of operations starting with node $eu \triangleq e.to$, the successor of uh along edge e.

3.3.3 Buffer

AJupiter maintains buffers (i.e., sequences) of operations of type *Op*. $xForm(op, ops)$ of module $Buffer$ (Fig.12) transforms an operation *op* against a buffer *ops* of operations (see Fig.13). It utilizes $xFormOpOps$ (op, ops) and $xFormOpsOp(ops, op)$ to obtain the last transformed operation *xop* and the transformed buffer *xops*, respectively. Specifically, $xFormOpOps$ returns the sequence of intermediate transformed operations, the last one of which is the desired *xop*.

1) If *ops* is empty, $\langle op \rangle$ is returned.

2) Otherwise, it prepends *op* to the resulting sequence obtained by recursively transforming $OT(op, Head(ops))$ against the tail $Tail(ops)$ of *ops*.

It also facilitates $xFormOpsOp$ to generate *xops* by transforming each operation in *ops* against the corresponding one in $opX \triangleq xFormOpOps(op, ops)$. Finally, $xFormShift(op, ops, shift)$ transforms *op* against the subsequence of *ops* obtained by shifting the first $shift$ operations out of *ops*.

4 Jupiter Protocols

In this section, we formally specify Jupiter protocols in TLA$^+$, including AbsJupiter that we propose as an abstract solution. We focus on when and how OTs are performed and on the data structures supporting OTs. As running examples, we will illustrate the behaviors of client c_3 in different Jupiter protocols under the schedule of Fig.1.

4.1 AbsJupiter

In AbsJupiter (Fig.14), each replica r maintains a set $copss[r]$ of context-based operations. The operator

MODULE *Buffer*

$xFormOpOps(op, ops) \triangleq$ Transform *op* against *ops*
  IF $ops = \langle\rangle$ THEN $\langle op \rangle$ and return intermediate transformed operations.
  ELSE $\langle op \rangle \circ xFormOpOps(OT(op, Head(ops)), Tail(ops))$
$xFormOpsOp(ops, op) \triangleq$ Transform *ops* against *op* and return the transformed *ops*.
  LET $opX \triangleq xFormOpOps(op, ops)$
  IN $[i \in 1 \,..\, Len(ops) \mapsto OT(ops[i], opX[i])]$
$xForm(op, ops) \triangleq$
  $[xop \mapsto Last(xFormOpOps(op, ops)), xops \mapsto xFormOpsOp(ops, op)]$
$xFormShift(op, ops, shift) \triangleq xForm(op, SubSeq(ops, shift + 1, Len(ops)))$

Fig.12. TLA$^+$ module $Buffer$.

$Perform(r, cop)$ calls $xForm(r, cop)$ of module *Set* to transform *cop* in $copss[r]$. The transformed operation $xform.xcop.op$ is applied to $list[r]$ and $copss[r]$ is updated to $xform.xcopss$.

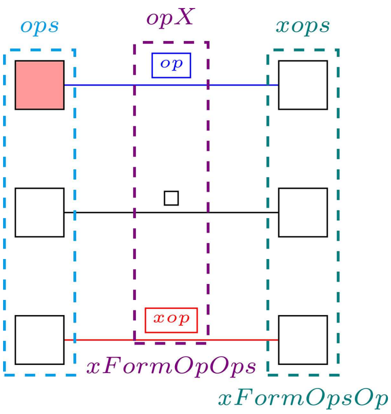


Fig.13. Illustration of *xForm* of module *Buffer*.

In $Do(c, op)$, the client c first wraps *op* into a context-based operation *cop* by attaching *oid* and $ctx = ds[c]$ to it. Then it updates $copss[c]$ to include *cop*, applies *op* to $list[c]$, and sends *cop* to the server. When the server receives a context-based operation *cop* from client c, it calls $Perform(Server, cop)$ and then broadcasts *cop* to other clients except c (see $SRev(cop)$). In $Rev(c, cop)$, client c just calls $Perform(c, cop)$.

Thanks to the mathematical set it uses, AbsJupiter is abstract from implementations with concrete data structures. As shown in Section 5, it embraces the other three Jupiter protocols as refinements.

Example 4 (*Illustration of AbsJupiter*). We illustrate client c_3 in AbsJupiter under the schedule of Fig.1 (see also Fig.15(a)). For convenience, we denote, for instance, an operation o_3 with context $\{o_1, o_2, o_4\}$ by $o_3\{o_1, o_2, o_4\}$.

After receiving and applying $o_1\{\,\}$ ($\text{INS}(x, 1)$) of client c_1 from the server, client c_3 generates o_4 ($\text{INS}(b, 2)$). It wraps o_4 into a context-based operation $o_4\{o_1\}$, adds $o_4\{o_1\}$ to $copss[c_3] = \{o_1\{\,\}\}$, applies o_4 locally, and then sends $o_4\{o_1\}$ to the server.

Next, client c_3 receives $o_2\{o_1\}$ ($\text{DEL}(1)$) of client c_1 from the server. By $xForm(c_3, o_2\{o_1\})$, it transforms $o_2\{o_1\}$ against the set of context-based operations in $copss[c_3] = \{o_1\{\,\}, o_4\{o_1\}\}$. Since o_4 is the only concurrent operation with o_2 in $copss[c_3]$, $o_2\{o_1\}$ and $o_4\{o_1\}$ are transformed against each other. As a result, the new context-based operations $o_2\{o_1, o_4\}$ ($\text{DEL}(1)$) and $o_4\{o_1, o_2\}$ ($\text{INS}(b, 1)$) are added into $copss[c_3]$. The transformed operation $\text{DEL}(1)$ is applied locally.

Finally, client c_3 receives $o_3\{o_1\}$ ($\text{INS}(a, 1)$) of client c_2 from the server. By $xForm(c_3, o_3\{o_1\})$, it transforms $o_3\{o_1\}$ against the set of context-based operations in $copss[c_3] = \{o_1\{\,\}, o_4\{o_1\}, o_2\{o_1\}, o_4\{o_1, o_2\}, o_2\{o_1, o_4\}\}$. The set of concurrent operations with o_3 in $copss[c_3]$ is calculated as $\{o_1, o_2, o_4\} \setminus \{o_1\} = \{o_2, o_4\}$. Since o_2 precedes o_4 in the SO order according to $serial[c_3] = \langle o_1, o_2 \rangle$, $o_3\{o_1\}$ is first transformed with $o_2\{o_1\}$, yielding $o_3\{o_1, o_2\}$ ($\text{INS}(a, 1)$) and $o_2\{o_1, o_3\}$ ($\text{DEL}(2)$). Then, $o_3\{o_1, o_2\}$ is transformed with $o_4\{o_1, o_2\}$ ($\text{INS}(b, 1)$), yielding $o_3\{o_1, o_2, o_4\}$ ($\text{INS}(a, 2)$) and $o_4\{o_1, o_2, o_3\}$ ($\text{INS}(b, 1)$). At last, c_3 applies the transformed operation $\text{INS}(a, 2)$ locally, obtaining the list *ba*.

4.2 CJupiter

In CJupiter (Fig.16), each replica r maintains an n-ary digraph $css[r]$ (initially *EmptyGraph*), a digraph where the outdegree of each node can be at most n (see module CJupiter). In $Do(c, op)$, the client c first wraps *op* into a context-based operation *cop*. Then it applies *op* to $list[c]$, inserts an edge labeled by *cop* from the node $ds[c]$ in $css[c]$, and sends *cop* to the server. The definitions of *Rev* and *SRev* of CJupiter are the same as those of AbsJupiter, ex-

MODULE *AbsJupiter*

VARIABLES *copss* $copss[r]$: the set of context-based operations maintained at replica r

$Perform(r, cop) \triangleq$ LET $xform \triangleq xForm(r, cop)$ $xform : [xcop, xcopss]$
IN $\wedge\ copss' = [copss$ EXCEPT $![r] = xform.xcopss]$
$\wedge$ apply $xform.xcop.op$ to $list[r]$

$Do(c, op) \triangleq$ LET $cop \triangleq [op \mapsto op, oid \mapsto [c \mapsto c, seq \mapsto cseq[c]], ctx \mapsto ds[c]]$
IN $\wedge\ copss = [copss$ EXCEPT $![c] = @ \cup \{cop\}]$
$\wedge$ apply *op* to $list[c]$; send *cop* to the *Server*

$Rev(c, cop) \triangleq Perform(c, cop)$

$SRev(cop) \triangleq \wedge\ Perform(Server, cop)$
$\wedge$ broadcast *cop* to clients other than $ClientOf(cop)$

Fig.14. TLA$^+$ module *AbsJupiter*.

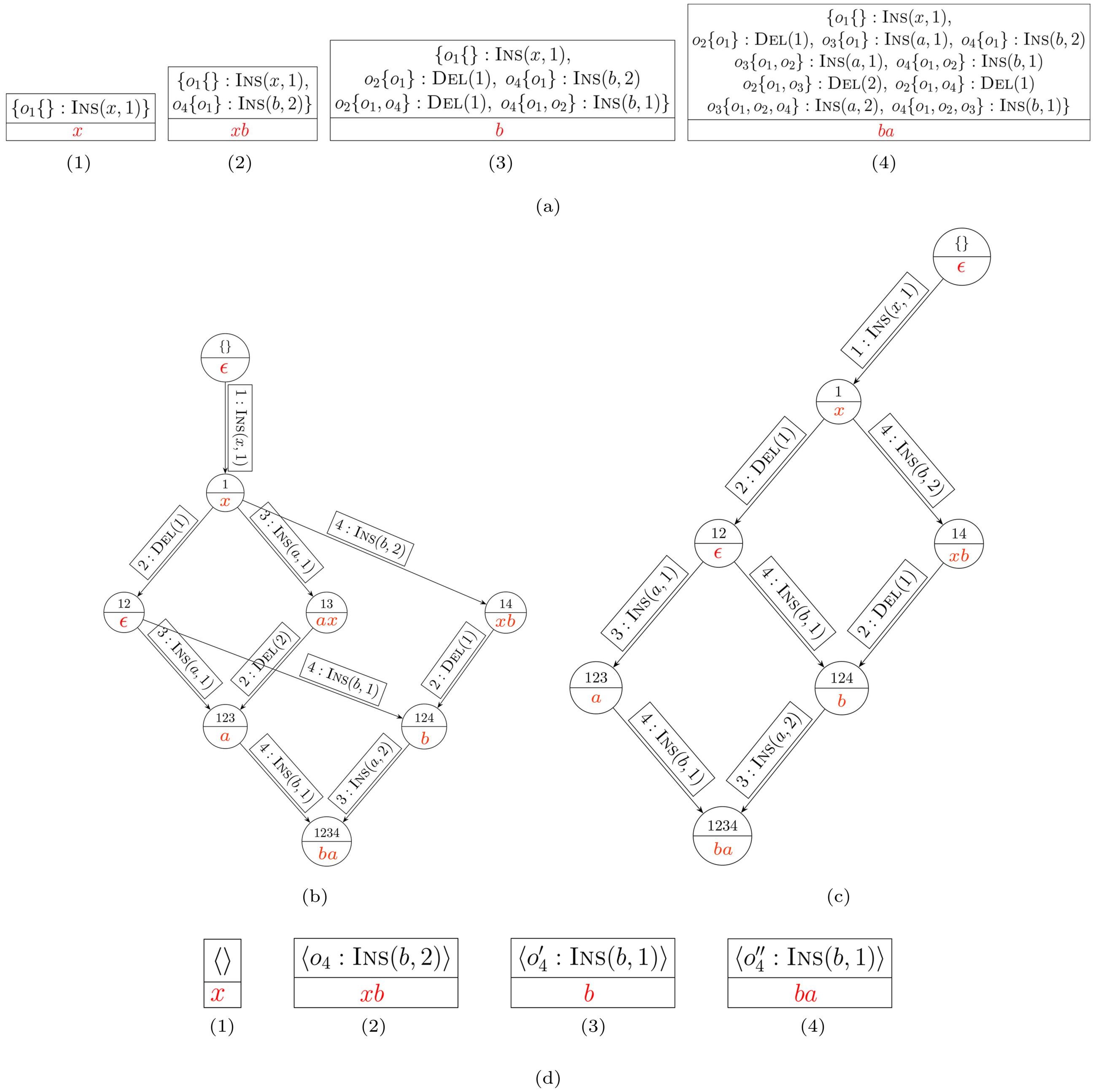


Fig.15. Illustration of client c_3 in Jupiter protocols under the schedule of Fig.1. (a) AbsJupiter. (b) CJupiter. (c) XJupiter. (d) AJupiter.

cept that $xForm(NextEdge, r, cop, css[r])$ of module $Digraph$ is called by replica r to transform cop against a sequence of context-based operations with cop along a path in digraph $css[r]$. The next edge from a given node chosen in $NextEdge$ is the first one in terms of SO according to the serial view $serial[r]$ of r. The intermediate $xform.xg$ produced in $xForm$ is integrated into $css[r]$ and the transformed operation $xform.xcop.op$ is applied to $list[r]$.

It is remarkable that although $(n+1)$ n-ary digraphs are maintained by CJupiter, they are (eventually) all the same. In other words, at a high level, CJupiter maintains only a single n-ary digraph, which contains exactly all replica states across the system [6]. This makes it feasible to reason about global properties like weak list specification [2,6].

Example 5 (*Illustration of CJupiter, Adapted from* [6]). We illustrate client c_3 in CJupiter under the schedule of Fig.1 (also see Fig.15(b)). For convenience, we denote, for instance, a node v with document state

```
MODULE CJupiter
VARIABLES css    css[r]: the n-ary digraph maintained at replica r

NextEdge(r, u, g) ≜ CHOOSE e ∈ g.edge : ∧ e.from = u
                                        ∧ ∀ ue ∈ g.edge \ {e} :
                        (ue.from = u) ⇒ so(e.cop.oid, ue.cop.oid, serial[r])
Perform(r, cop) ≜ LET xform ≜ xForm(NextEdge, r, cop, css[r])
                  IN  ∧ css' = [css EXCEPT ![r] = @ ⊕ xform.xg]
                      ∧ apply xform.xcop.op to list[r]
Do(c, op) ≜ LET cop ≜ [op ↦ op, oid ↦ [c ↦ c, seq ↦ cseq[c]], ctx ↦ ds[c]]
                u ≜ ds[c]    v ≜ u ∪ {cop.oid}
            IN  ∧ css' = [css EXCEPT ![c] =
                         @ ⊕ [node ↦ {v},
                              edge ↦ {[from ↦ u, to ↦ v, cop ↦ cop]}]]
                ∧ apply op to list[c]; send cop to the Server
Rev(c, cop) ≜ Perform(c, cop)
SRev(cop) ≜ ∧ Perform(Server, cop)
            ∧ broadcast cop to clients other than ClientOf(cop)
```

Fig.16. TLA$^+$ module $CJupiter$.

$\{o_1, o_4\}$ by v_{14}.

After receiving and applying $o_1\{\}$ of client c_1 redirected by the server, client c_3 generates o_4 ($\textsc{Ins}(b,2)$). It wraps o_4 into a context-based operation $o_4\{o_1\}$, links a new node v_{14} to v_1 via an edge labeled by $o_4\{o_1\}$, and then sends $o_4\{o_1\}$ to the server.

Next, client c_3 receives $o_2\{o_1\}$ ($\textsc{Del}(1)$) of client c_1 from the server. The context of $o_2\{o_1\}$ matches node v_1. By $xForm$, $o_2\{o_1\}$ and $o_4\{o_1\}$ are transformed against each other. Node v_{124} is created and is linked to v_{12} and v_{14} via the edges labeled with $o_4\{o_1, o_2\}$ ($\textsc{Ins}(b,1)$) and $o_2\{o_1, o_4\}$ ($\textsc{Del}(1)$), respectively.

Finally, client c_3 receives $o_3\{o_1\}$ ($\textsc{Ins}(a,1)$) of client c_2 from the server. The context of $o_3\{o_1\}$ matches node v_1. By $xForm$, $o_3\{o_1\}$ will be transformed with the operation sequence consisting of operations along the "first" (in terms of SO with $serial[c_3] = \langle o_1, o_2\rangle$) edges from v_1 to v_{124}. Specifically, $o_3\{o_1\}$ is first transformed with $o_2\{o_1\}$. Then, $o_3\{o_1, o_2\}$ ($\textsc{Ins}(a,1)$) is transformed with $o_4\{o_1, o_2\}$ ($\textsc{Ins}(b,1)$), yielding v_{1234}, $o_3\{o_1, o_2, o_4\}$ ($\textsc{Ins}(a,2)$), and $o_4\{o_1, o_2, o_3\}$ ($\textsc{Ins}(b,1)$). Client c_3 applies $\textsc{Ins}(a,2)$, obtaining list ba.

4.3 XJupiter

XJupiter (Fig.17) uses 2-dimentional (2D) digraphs where the outdegree of each node is at most 2. Each client c maintains a single 2D digraph $c2ss[c]$, and the server maintains n 2D digraphs, one digraph $s2ss[c]$ per client c. Conceptually, a 2D digraph, either $c2ss[c]$ or $s2ss[c]$, has two dimensions: a local dimension for storing operations generated by c and a remote dimension for storing operations generated by other clients.

In $Do(c, op)$, the client c first wraps op into a context-based operation cop by attaching oid and $ctx = ds[c]$ to it. Then it applies op to $list[c]$, inserts an edge labeled by cop from node $ds[c]$ in $c2ss[c]$ along the local dimension, and sends cop to the server.

When the server receives a context-based operation cop from client c, it transforms cop against the context-based operations along the remote dimension from node $u \triangleq cop.ctx$ to $ds[Server]$ in $s2ss[c]$. In $SRev(cop)$, this is done in $xForm(NextEdge, Server, cop, s2ss[c])$ of module $Digraph$, where $NextEdge$ returns the unique outgoing edge of a given node. Then, the transformed operation $xform.xcop.op$ is applied to $list[Server]$, $s2ss[c]$ is updated to integrate $xform.xg$, and $xform.lg$ is inserted to the remote dimension of each digraph $s2ss[cl \neq c]$. Finally, the server broadcasts the transformed context-based operation $xform.xcop$ to other clients except c.

When client c receives a context-based operation cop from the server, it calls $xForm(NextEdge, c, cop, c2ss[c])$ of module $Digraph$ to transform cop against the operations along the local dimension from node $u \triangleq cop.ctx$ to $ds[c]$ in $c2ss[c]$. The intermediate $xform.xg$ is integrated into $c2ss[c]$ and the transformed operation $xform.xcop.op$ is applied to $list[c]$.

Since the transformed context-based operations are broadcast by the server in XJupiter, XJupiter is slightly optimized in implementation at clients with respect to CJupiter, by eliminating redundant OTs that have already been performed at the server [6]. More importantly, this improvement makes it possible to reduce n-ary digraphs to 2D-digraphs.

```
──────────────────────────── MODULE XJupiter ────────────────────────────
VARIABLES c2ss,   c2ss[c]: the 2D digraph maintained at client c
          s2ss    s2ss[c]: the 2D digraph maintained by the Server for client c
─────────────────────────────────────────────────────────────────────────
NextEdge(_, u, g) ≜ CHOOSE e ∈ g.edge : e.from = u
Do(c, op) ≜ LET cop ≜ [op ↦ op, oid ↦ [c ↦ c, seq ↦ cseq[c]], ctx ↦ ds[c]]
                 u ≜ ds[c]     v ≜ u ∪ {cop.oid}
            IN   ∧ c2ss' = [c2ss EXCEPT ![c] =
                       @ ⊕ [node ↦ {v},
                            edge ↦ {[from ↦ u, to ↦ v, cop ↦ cop]}]]
                 ∧ apply op to list[c]; send cop to the Server
Rev(c, cop) ≜ LET xform ≜ xForm(NextEdge, c, cop, c2ss[c])
                  xform: [xcop, xg, lg]
              IN  ∧ c2ss' = [c2ss EXCEPT ![c] = @ ⊕ xform.xg]
                  ∧ apply xform.xcop.op to list[c]
SRev(cop) ≜
    LET c ≜ ClientOf(cop)
    xform ≜ xForm(NextEdge, Server, cop, s2ss[c])  xform: [xcop, xg, lg]
    IN  ∧ s2ss' = [cl ∈ Client ↦ IF cl = c THEN s2ss[cl] ⊕ xform.xg
                                           ELSE s2ss[cl] ⊕ xform.lg]
        ∧ apply xform.xcop.op to list[Server]
        ∧ broadcast the transformed operation xform.xcop to clients other than c
─────────────────────────────────────────────────────────────────────────
```

Fig.17. TLA$^+$ module *XJupiter*.

Example 6 (*Illustration of XJupiter. Adapted from* [6]). We illustrate client c_3, as well as *Server*, in XJupiter under the schedule of Fig.1 (see Fig.18 and Fig.15(c)). Client c_3 in XJupiter behaves similarly as it does in CJupiter, when it receives o_1 of client c_1, o_4 generated by itself, and o_2 of client c_1.

We now explain what c_3 does when it receives o_3 of client c_2 redirected by the server. Client c_2 has propagated its operation $o_3\{o_1\}$ (INS$(a, 1)$) to the server. At the server, $o_3\{o_1\}$ was transformed with $o_2\{o_1\}$

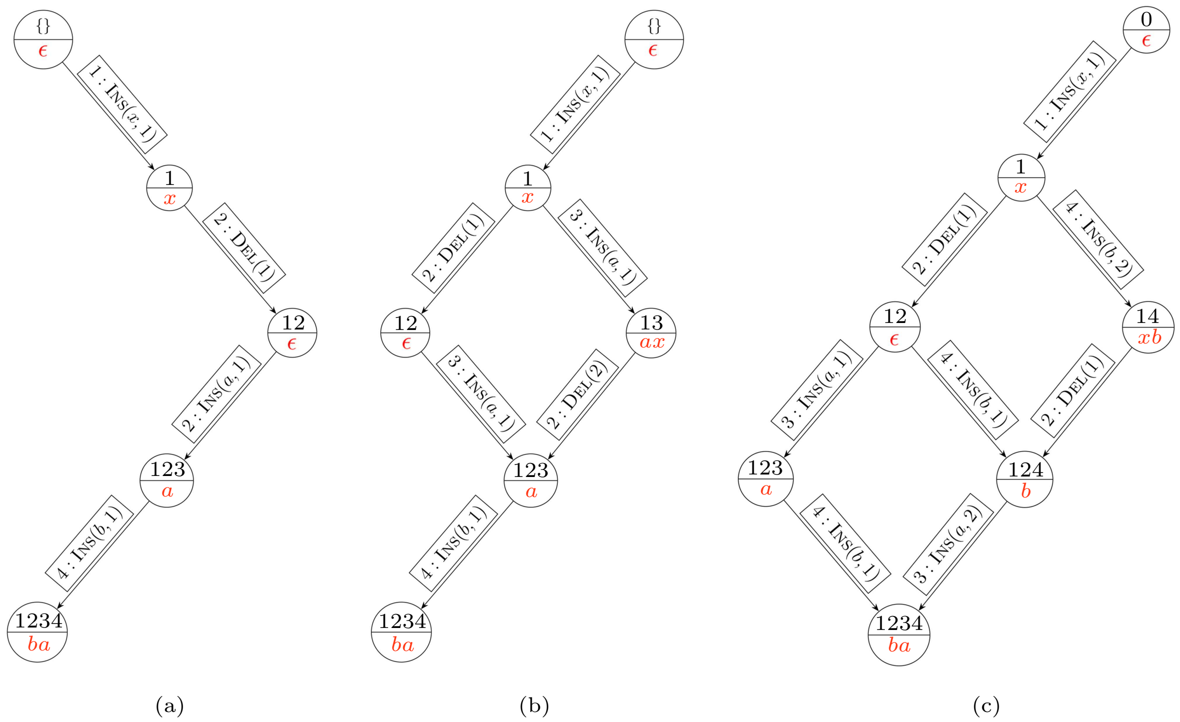


Fig.18. Illustration of the server in XJupiter under the schedule of Fig.1. (a) $s2ss[c_1]$. (b) $s2ss[c_2]$. (c) $s2ss[c_3]$. ↘: local dimension; ↙: remote dimension.

(Del(1)) along the remote dimension in $s2ss[c_2]$, obtaining $o_3\{o_1, o_2\}$ (Ins$(a, 1)$). Besides being stored in $s2ss[c_1]$ and $s2ss[c_3]$, $o_3\{o_1, o_2\}$ (instead of $o_3\{o_1\}$ that the server receives) is redirected by the server to clients c_1 and c_3. At client c_3, the context of $o_3\{o_1, o_2\}$ matches node v_{12} in $c2ss[c_3]$. By $xForm$ of $Digraph$, $o_3\{o_1, o_2\}$ should be transformed against the operations along the local dimension (in the southeast arrow "$\searrow$" in Fig.15(c)) from node v_{12} in $c2ss[c_3]$. In this example, $o_3\{o_1, o_2\}$ is transformed with $o_4\{o_1, o_2\}$ (Ins$(b, 1)$), yielding v_{1234}, $o_3\{o_1, o_2, o_4\}$ (Ins$(a, 2)$), and $o_4\{o_1, o_2, o_3\}$ (Ins$(b, 1)$). Finally, client c_3 applies Ins$(a, 2)$, obtaining the list ba.

4.4 AJupiter

In AJupiter (Fig.19), each client c maintains a buffer $cbuf[c]$ for storing the operations (maybe transformed) it generates, and a counter $crec[c]$ counting the number of operations it has received from the server since the last time it generated an operation and sent a message. Similarly, the server maintains for each client c a buffer $sbuf[c]$ for storing the (transformed) operations generated by other clients except c, and a counter $srec[c]$ counting the number of operations the server has received from client c since the last time an operation which was generated by other clients except c was transformed at the server and a message was broadcast.

The counters (i.e., $crec[c]$ and $srec[c]$) are piggybacked in the ack field in messages $AJMsg$ telling the other side how many new messages have been received since the last time a message was sent (see module $AJupiter$). When a client c receives a message m of form $[ack \mapsto srec[c], op \mapsto xop]$ broadcast by $Server$, it knows that op is generated by another client and more importantly that the set of operations against which op has been transformed at $Server$ contains the first ack operations in $cbuf[c]$. Thus, in $Rev(c, m)$, client c calls $xFormShift(m.op, cbuf[c], m.ack)$ of module $Buffer$ to transform op against the subsequence of operations obtained by shifting the first $m.ack$ operations out of $cbuf[c]$. Similarly, when $Server$ receives a message m of form $[c \mapsto c, ack \mapsto crec[c], op \mapsto op]$ from client c, it knows that among the (transformed) operations in $sbuf[c]$ generated by other clients except c, the first ack operations have been broadcast to c and have been transformed at c before op was generated. Thus, in $SRev(m)$, $Server$ calls $xFormShift(m.op, sbuf[c], m.ack)$ of module $Buffer$ to transform op against the subsequence of operations obtained by shifting the first $m.ack$ operations out of $sbuf[c]$. The transformed operation xop will be appended to other $sbuf[cl]$ for clients $cl \neq c$. Finally, $Server$ sends the transformed operation xop along with

```
───────────────────────────── MODULE AJupiter ─────────────────────────────
VARIABLES cbuf, crec, sbuf, srec
AJMsg ≜ [c : Client, ack : Nat, op : Op ∪ {Nop}] ∪   from client c to Server
        [ack : Nat, op : Op ∪ {Nop}]   from Server to clients
────────────────────────────────────────────────────────────────────────────
Do(c, op) ≜ ∧ cbuf' = [cbuf EXCEPT ![c] = Append(@, op)]
            ∧ crec' = [crec EXCEPT ![c] = 0]
            ∧  apply op to list[c]
            ∧  send [c ↦ c, ack ↦ crec[c], op ↦ op] to the Server
Rev(c, m) ≜ LET xform ≜ xFormShift(m.op, cbuf[c], m.ack)
                  xform : [xop, xops]
            IN    ∧ cbuf' = [cbuf EXCEPT ![c] = xform.xops]
                  ∧ crec' = [crec EXCEPT ![c] = @ + 1]
                  ∧  apply xform.xop to list[c]
SRev(m) ≜ LET c ≜ m.c
              xform ≜ xFormShift(m.op, sbuf[c], m.ack)   xform : [xop, xops]
                xop ≜ xform.xop
          IN    ∧ srec' = [cl ∈ Client ↦
                              IF cl = c THEN srec[cl] + 1 ELSE 0]
                ∧ sbuf' = [cl ∈ Client ↦
                              IF cl = c THEN xform.xops
                                        ELSE Append(sbuf[cl], xop)]
                ∧  apply xop to list[Server]
                ∧  send [ack ↦ srec[cl], op ↦ xop] to client cl ≠ c
────────────────────────────────────────────────────────────────────────────
```

Fig.19. TLA$^+$ module $AJupiter$.

$srec[cl]$ to client $cl \neq c$.

By maintaining only 1D buffers and discarding/shifting obsolete operations whenever possible, AJupiter is the most efficient one among these four Jupiter protocols.

Example 7 (*Illustration of AJupiter*). We illustrate client c_3 in AJupiter under the schedule of Fig.1 (see also Fig.15(d)).

First, when client c_3 receives o_1 ($\textsc{Ins}(x, 1)$) of client c_1 from the server, its buffer $cbuf[c_3]$ is empty. Therefore, in *Rec*, it simply increases $crec[c_3]$ by 1 and applies $\textsc{Ins}(x, 1)$ locally.

Next, client c_3 generates o_4 ($\textsc{Ins}(b, 2)$). In *Do*, it appends o_4 to its currently empty buffer $cbuf[c_3]$, resets $crec[c_3]$ to 0, applies o_4 locally, and sends o_4 with $ack = 1$ to the server.

Then, client c_3 receives o_2 ($\textsc{Del}(1)$) with $ack = 0$ of client c_1 from the server. By $xForm$ of $Buffer$, o_2 ($\textsc{Del}(1)$) is transformed against o_4 ($\textsc{Ins}(b, 2)$) in buffer $cbuf[c_3]$. The transformed operation $OT(o_2, o_4) = \textsc{Del}(1)$ is applied locally, and o_4 in buffer $cbuf[c_3]$ is transformed into $OT(o_4, o_2) = Ins(b, 1)$.

Finally, client c_3 receives transformed o_3 ($\textsc{Ins}(a, 1)$ which happens to be unchanged) with $ack = 0$ of client c_2 from the server. By $xForm$ of $Buffer$, o_3 ($\textsc{Del}(1)$) is transformed against o_4 (which is now $\textsc{Ins}(b, 1)$) in buffer $cbuf[c_3]$. The transformed operation $OT(o_3, o_4) = \textsc{Del}(2)$ is applied locally, obtaining the list *ba*. Meanwhile, o_4 in buffer $cbuf[c_3]$ is transformed into $OT(o_4, o_3) = \textsc{Ins}(b, 1)$.

5 Refinement

The OT behaviors (namely, when and how to perform OTs) of four Jupiter protocols are essentially the same under the same schedule of actions of *Do*, *Rev*, and *SRev*. The main difference lies in the data structures they use to support OTs (see Fig.20). Specifically, AbsJupiter maintains sets of context-based operations. CJupiter organizes these context-based operations into n-ary digraphs, by grouping the ones with the same context. Since the transformed context-based operations are broadcast by the server in XJupiter, XJupiter is slightly optimized in implementation at clients by eliminating redundant OTs that have already been performed at the server [6]. XJupiter synchronizes each client with its counterpart at the server, where 2D digraphs that distinguish the local dimension from the remote dimension are sufficient. In AJupiter, each client maintains only the local dimension for operations it generates, and the remote dimension for operations generated by other clients is maintained by its counterpart at the server. Thus, 2D digraphs can be reduced to 1D buffers. In this section, we establish the (data) refinement relation [12–14] among these Jupiter protocols. Specifically, we show that AJupiter is a refinement of XJupiter, XJupiter is a refinement of CJupiter, and CJupiter is a refinement of AbsJupiter, by defining (data) refinement mappings to simulate the data structure of one Jupiter protocol using that of another Jupiter protocol. In the following, we focus on the refinement mappings for data structures mentioned above, and omit details for other variables.

5.1 CJupiter Refines AbsJupiter

The set $copss[r]$ of context-based operations maintained at replica r in AbsJupiter has been organized into an n-ary digraph $css[r]$ in CJupiter, by grouping the ones with the same context. Therefore, the refinement mapping from CJupiter to AbsJupiter only needs to simulate $copss[r]$ in AbsJupiter by extracting the context-based operations associated with the edges of $css[r]$ in CJupiter (see its definition in module *CJupiterImplAbsJupiter* (Fig.21)).

5.2 XJupiter Refines CJupiter

The refinement mapping from XJupiter to CJupiter defined in module *XJupiterImplCJupiter* (Fig.22) simulates, for each replica, the n-ary digraph in CJupiter using the 2D digraph(s) in XJupiter.

At the server side, XJupiter has decomposed the single n-ary digraph $css[Server]$ in CJupiter into n 2D digraphs, one $s2ss[c]$ for each client c. Thus, the refinement mapping simulates $css[Server]$ by taking the union of these $s2ss[c]$ for all $c \in Client$. Conceptually, this can be expressed in TLA$^+$ as (not syntactically correct):

$$css[Server] \leftarrow SetReduce(\oplus, Range(s2ss), EmptyGraph),$$

where $Range(s2ss)$ is the set of $s2ss[c]$ for all c, and $SetReduce$ combines $Range(s2ss)$ into one using $\oplus$ with an empty digraph as the initial value.

The server in XJupiter broadcasts the transformed operation $xform.xcop$ (instead of cop that it receives) to clients. Thus, the clients can skip the OTs transforming cop to $xform.xcop$ performed at the server. To simulate the n-ary digraph $css[c]$ at client c in

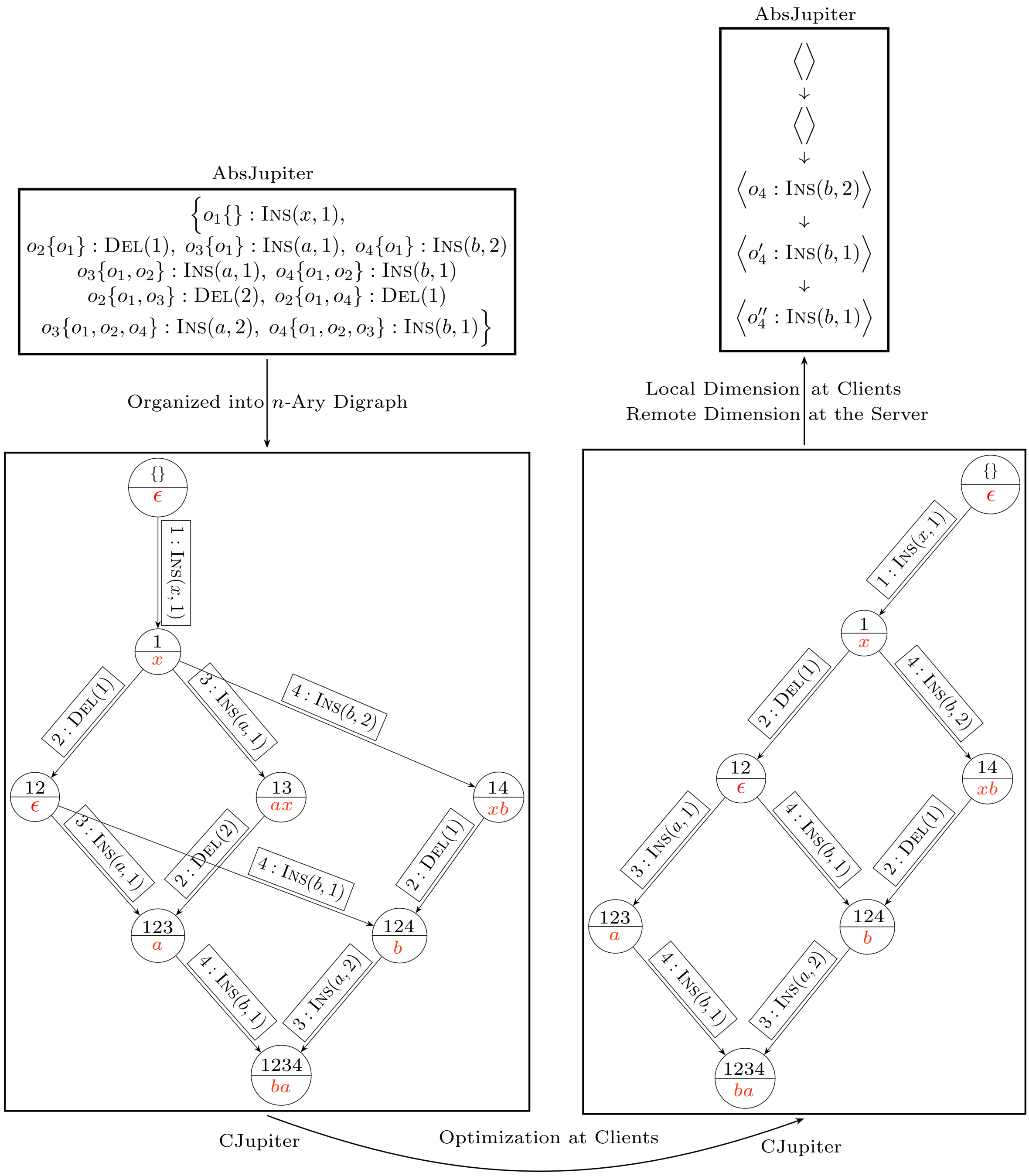


Fig.20. Illustration of the data refinement relation among Jupiter protocols (taking client c_3 in Fig.1 as an example). First, context-based operations with the same context of AbsJupiter are connected to the same node in the digraph of CJupiter. Second, the redundant OTs performed at the server have been optimized away from the digraph of XJupiter. Finally, only the transformed operations along the local dimension of the digraph of XJupiter are kept in the buffer of AJupiter.

MODULE *CJupiterImplAbsJupiter*

EXTENDS *CJupiter*

$AbsJ \triangleq$ INSTANCE *AbsJupiter*

WITH $copss \leftarrow [r \in Replica \mapsto \{e.cop : e \in css[r].edge\}]$

Fig.21. TLA$^+$ module *CJupiterImplAbsJupiter*.

MODULE $XJupiterImplCjupiter$

EXTENDS $XJupiter$

We have omitted the history variables for recording serial views.

VARIABLES $op2ss$, a function mapping an operation (identifier) to the 2D digraph produced during its transformation at the server

$c2ssX$ $c2ssX[c]$: 2D digraph that has been skipped by client c

$$
\begin{aligned}
InitImpl \triangleq\ & \land Init \\
& \land \text{ on history variables for serial views} \\
& \land op2ss = \langle\rangle \text{ the empty function expressed in TLA}^+ \\
& \land c2ssX = [c \in Client \mapsto EmptyGraph] \\
DoImpl(c, op) \triangleq\ & \land Do(c, op) \\
& \land \text{ on history variables for serial views} \\
RevImpl(c, cop) \triangleq\ & \land Rev(c, cop) \\
& \land \text{ on history variables for serial views} \\
& \land c2ssX' = [c2ssX \text{ EXCEPT } ![c] = @ \oplus op2ss[cop.oid]] \\
SRevImpl(cop) \triangleq\ & \land SRev(cop) \\
& \land \text{ on history variables for serial views} \\
& \land \text{LET } xform \triangleq xForm(NextEdge, Server, cop, s2ss[ClientOf(cop)]) \\
& \quad \text{IN } op2ss' = op2ss \mathbin{@@} (cop.oid :> xform.xg)
\end{aligned}
$$

$$
\begin{aligned}
CJ \triangleq \text{ INSTANCE } CJupiter \text{ WITH } css \leftarrow [r \in Replica \mapsto\ & \text{IF } r = Server \text{ THEN } SetReduce(\oplus, Range(s2ss), EmptyGraph) \\
& \text{ELSE } c2ss[r] \oplus c2ssX[r]]
\end{aligned}
$$

Fig.22. TLA$^+$ module $XJupiterImplCJupiter$.

CJupiter using the 2D digraph $c2ss[c]$ in XJupiter, we need to complement $c2ss[c]$ with those OTs skipped by XJupiter. To this end, we introduce two history variables in XJupiterImplCJupiter to record OTs. The variable $op2ss$ is a function mapping an operation (identifier) to the extra 2D digraph produced during its transformation at the server. When an operation cop is transformed at the server, the new mapping $cop.oid :> xform.xg$ is added to $op2ss$ (see $SRevImpl(cop)$). When client c receives the transformed operation $xform.xcop$ broadcast by the server, it accumulates this extra 2D digraph $op2ss[cop.oid]$ into $c2ssX[c]$, the overall 2D digraph that has been skipped by client c (see $RevImpl(c, cop)$). Thus, for client c, the simulation between $css[c]$ and $c2ss[c]$ can be (conceptually) expressed as $css[c] \leftarrow c2ss[c] \oplus c2ssX[c]$.

5.3 AJupiter Refines XJupiter

AJupiter uses 1D buffers to replace 2D digraphs in XJupiter, by keeping only the latest operation sequences that should participate in further OTs and discarding the old ones and intermediate transformed operations. Therefore, the refinement mapping needs to reconstruct these 2D digraphs in XJupiter from the OTs performed on 1D buffers in AJupiter. To this end, we introduce two history variables $c2ss$ and $s2ss$ in AJupiterImplXJupiter (Fig.23) which are to simulate $c2ss$ and $s2ss$ in XJupiter, respectively. They are supposed to be updated in accordance with $cbuf$ and $sbuf$ of AJupiter. Specifically, in $DoImpl(c, op)$, the generated operation op is wrapped as a context-based operation cop and added to $c2ss[c]$ as in XJupiter; besides it is stored in $cbuf[c]$ as in AJupiter (not shown here). In $RevImpl(c, m)$ and $SRevImpl(m)$, $xFormCopCopsShift$ behaves as $xFormShift$ and $xFormOpOps$ used in AJupiter, except that the former performs COTs on context-based operations and stores intermediate nodes and edges produced during COTs into $c2ss[c]$ and $s2ss$ as in XJupiter, respectively.

6 Model Checking Results

We first present the model checking results of verifying the refinement relation among Jupiter protocols. Thanks to the refinement relation, we then only need to verify AbsJupiter with respect to desired properties to ensure the correctness of all Jupiter protocols.

Verification by model checking is conducted by TLC [16] (implemented in the TLA$^+$ Toolbox of version 1.5.7), a model checker for TLA$^+$, on a 2.40 GHz 6-core machine with 64 GB RAM. For each group of model checking experiments, we vary the number of clients

```
──────────────── MODULE AJupiterImplXJupiter ────────────────
EXTENDS AJupiter
 We have omitted the history variables for recording operation contexts.
VARIABLES c2ss, s2ss
─────────────────────────────────────────────────────────────
InitImpl ≜ ∧ Init
           ∧  on history variables for operation contexts
           ∧ c2ss = [c ∈ Client ↦ EmptyGraph]
           ∧ s2ss = [c ∈ Client ↦ EmptyGraph]
DoImpl(c, op) ≜ ∧ Do(c, op)
                ∧  on history variables for operation contexts
                ∧ LET cop ≜ [op ↦ op,
                             oid ↦ [c ↦ c, seq ↦ cseq[c]], ctx ↦ ds[c]]
                    IN  c2ss′ = [c2ss EXCEPT ![c] =
                      @ ⊕ [node ↦ {ds′[c]},
                           edge ↦ {[from ↦ ds[c], to ↦ ds′[c], cop ↦ cop]}]]
RevImpl(c, m) ≜ ∧ Rev(c, m)
                ∧  on history variables for operation contexts
                ∧ LET xform ≜ xFormCopCopsShift(m.cop, cbuf[c], m.ack)
                    IN  c2ss′ = [c2ss EXCEPT ![c] = @ ⊕ xform.xg]
SRevImpl(m) ≜ ∧ SRev(m)
              ∧  on history variables for operation contexts
              ∧ LET c ≜ ClientOf(m.cop)
                    xform ≜ xFormCopCopsShift(m.cop, sbuf[c], m.ack)
                  IN  s2ss′ = [cl ∈ Client ↦
                                  IF cl = c THEN s2ss[cl] ⊕ xform.xg
                                            ELSE s2ss[cl] ⊕ xform.lg]
─────────────────────────────────────────────────────────────
XJ ≜ INSTANCE XJupiter WITH c2ss ← c2ss, s2ss ← s2ss
─────────────────────────────────────────────────────────────
```

Fig.23. TLA$^+$ module *AJupiterImplXJupiter*.

and the number of characters allowed to be inserted⑨. We use the symmetry set [15] for the set *Char* of characters. The initial lists on all replicas are empty. We use 10 threads and report the following statistics: the diameter of the reachable-state graph (i.e., the length of the longest behavior of protocol), the number of states TLC examines, the number of distinct states, and the checking time in hh:mm:ss.

6.1 Verifying Refinement Relation Among Jupiter Protocols

We verify the refinement mapping *AbsJ* from CJupiter to AbsJupiter defined in *CJupiterImplAbsJupiter* by checking that each behavior of CJupiter with variables substituted by *AbsJ* is a behavior allowed by AbsJupiter. The model checking results are shown in Table 3⑩. Similar results on verification of the refinement mappings defined in *XJupiterImplCJupiter* and *AJupiterImplXJupiter* are shown in Table 4 and Table 5, respectively.

6.2 Verifying Correctness of Jupiter Protocols

We present the model checking results of verifying that *AbsJupiter* satisfies the weak list specification WLSpec [2]. To express WLSpec in TLA$^+$, we introduce module *AbsJupiterH* (Fig.24) which extends AbsJupiter with a history variable *hlist* [13]. *AbsJupiterH* behaves exactly as *AbsJupiter*, except that it collects the new list state $list'[r]$ in each action into *hlist*. We

⑨The positive model checking results help to gain great confidence in the correctness of these Jupiter protocols and the refinement relation among them, given the empirical study [21] that "almost all failures (of 198 production failures in distributed data-intensive systems) require only three or fewer nodes to reproduce". In our experiments, with some configurations such as $(3, 2)$, we are able to explore the behaviors of the protocol with a diameter of the length greater than 30 and with more than 200 million states.

⑩In the table, "$\#x$" means "the number of x". Additionally, in a "starred" experiment, we exit TLC when the number of distinct states it examines reaches a threshold θ. This is supported by a TLA$^+$ Toolbox nightly build as of 01-28-2019 (at 05:56).

Table 3. Model Checking Results of Verifying That CJupiter Refines AbsJupiter

TLC Model (#Clients, #Chars)	Diameter	#States	#Distinct States	Checking Time (hh:mm:ss)
(1, 1)	5	7	6	00 : 00 : 00
(1, 2)	9	86	57	00 : 00 : 00
(1, 3)	13	1 696	1 014	00 : 00 : 01
(1, 4)	17	53 273	30 393	00 : 00 : 06
(2, 1)	10	71	53	00 : 00 : 01
(2, 2)	19	50 215	28 307	00 : 00 : 05
(2, 3)	28	150 627 005	75 726 121	04 : 37 : 36
(2, 4)	18	121 964 031	$\theta = 80\,000\,000^{\star}$	05 : 21 : 04
(3, 1)	17	2 785	1 288	00 : 00 : 01
(3, 2)	33	206 726 218	74 737 027	05 : 43 : 26
(3, 3)	18	139 943 577	$\theta = 80\,000\,000^{\star}$	05 : 18 : 57
(4, 1)	26	194 877	61 117	00 : 00 : 18
(4, 2)	21	177 451 069	$\theta = 80\,000\,000^{\star}$	06 : 12 : 48

Table 4. Model Checking Results of Verifying That XJupiter Refines CJupiter

TLC Model (#Clients, #Chars)	Diameter	#States	#Distinct States	Checking Time (hh:mm:ss)
(1, 1)	5	7	6	00 : 00 : 00
(1, 2)	9	86	57	00 : 00 : 00
(1, 3)	13	1 696	1 014	00 : 00 : 01
(1, 4)	17	53 273	30 393	00 : 00 : 07
(2, 1)	10	71	53	00 : 00 : 00
(2, 2)	19	50 215	28 307	00 : 00 : 07
(2, 3)	28	150 627 005	75 726 121	05 : 38 : 00
(2, 4)	19	122 113 291	$\theta = 80\,000\,000^{\star}$	08 : 01 : 35
(3, 1)	17	2 785	1 288	00 : 00 : 02
(3, 2)	33	206 726 218	74 737 027	08 : 50 : 40
(3, 3)	20	139 577 795	$\theta = 80\,000\,000^{\star}$	08 : 59 : 52
(4, 1)	26	194 877	61 117	00 : 00 : 30
(4, 2)	19	175 896 403	$\theta = 80\,000\,000^{\star}$	11 : 40 : 50

Table 5. Model Checking Results of Verifying That AJupiter Refines XJupiter

TLC Model (#Clients, #Chars)	Diameter	#States	#Distinct States	Checking Time (hh:mm:ss)
(1, 1)	5	7	6	00 : 00 : 01
(1, 2)	9	86	57	00 : 00 : 01
(1, 3)	13	1 696	1 014	00 : 00 : 01
(1, 4)	17	53 273	30 393	00 : 00 : 07
(2, 1)	10	71	53	00 : 00 : 00
(2, 2)	19	50 215	28 307	00 : 00 : 05
(2, 3)	28	150 627 005	75 726 121	04 : 23 : 52
(2, 4)	18	122 137 621	$\theta = 80\,000\,000^{\star}$	03 : 52 : 46
(3, 1)	17	2 785	1 288	00 : 00 : 01
(3, 2)	33	206 726 218	74 737 027	04 : 52 : 39
(3, 3)	18	139 823 551	$\theta = 80\,000\,000^{\star}$	04 : 48 : 23
(4, 1)	26	194 877	61 117	00 : 00 : 17
(4, 2)	21	176 794 063	$\theta = 80\,000\,000^{\star}$	03 : 49 : 58

check that WLSpec is an invariant of AbsJupiterH using TLC, and the model checking results are shown in Table 6.

7 Related Work

OT was pioneered by Sun and Ellis in 1989[1]. Though the idea of OT is simple, OT-based protocols are subtle and error-prone. For example, the dOPT protocol in [1] for P2P systems does not work in all cases[7,8]. Remarkably, after several failed attempts[8,9,22], it was shown impossible[10,11] to design OT functions (and thus OT-based protocols) for P2P systems for lists with signatures of *Ins* and *Del* as described in Subsection 2.3. In other words, extra

MODULE *AbsJupierH*

EXTENDS *AbsJupiter*
VARIABLE *hlist*

$InitH \triangleq Init \land hlist = \{\}$
$DoH(c) \triangleq Do(c) \land hlist' = hlist \cup \{list'[c]\}$
$RevH(c) \triangleq Rev(c) \land hlist' = hlist \cup \{list'[c]\}$
$SRevH \triangleq SRev \land hlist' = hlist \cup \{list'[Server]\}$

$WLSpec \triangleq \forall l1, l2 \in hlist : Compatible(l1, l2)$

Fig.24. TLA$^+$ module *AbsJupierH*.

Table 6. Model Checking Results of Verifying that AbsJupiter Satisfies $WLSpec$

TLC Model (#Clients, #Chars)	Diameter	#States	#Distinct States	Checking Time (hh:mm:ss)
(1, 1)	5	7	6	00 : 00 : 01
(1, 2)	9	86	57	00 : 00 : 01
(1, 3)	13	1 696	1 014	00 : 00 : 00
(1, 4)	17	53 273	30 393	00 : 00 : 04
(2, 1)	10	71	53	00 : 00 : 00
(2, 2)	19	50 215	28 307	00 : 00 : 03
(2, 3)	28	150 627 005	75 726 121	01 : 54 : 46
(2, 4)	20	153 275 009	$\theta = 100\,000\,000^\star$	03 : 54 : 49
(3, 1)	17	2 785	1 288	00 : 00 : 01
(3, 2)	33	206 726 218	74 737 027	02 : 46 : 02
(3, 3)	25	175 457 016	$\theta = 100\,000\,000^\star$	02 : 59 : 29
(4, 1)	26	194 877	61 117	00 : 00 : 09
(4, 2)	22	222 738 876	$\theta = 100\,000\,000^\star$	03 : 16 : 45

parameters are needed for *Ins* and *Del* operations [11]. On the other hand, researchers made efforts to gain a better understanding why some OT-based protocols work [4, 20].

The first Jupiter protocol appeared in 1995 [3] and is now used in many collaborative editors such as Google Docs⑪, Firepad, and SubEthaEdit. However, its original description involves only a single client. Based on the notion of COT Xu *et al.* [4] developed before [20], they reported a multi-client version of Jupiter, which we call XJupiter. XJupiter uses 2D digraphs to manage COTs. Independently, Attiya and Gotsman described another multi-client version of Jupiter, which we call AJupiter⑫. AJupiter relies on the acknowledgment mechanism and uses 1D buffers to manage OTs, thus reducing the metadata overhead. To facilitate the proof that XJupiter satisfies the weak list specification [2], Wei *et al.* [6] proposed CJupiter (Compact Jupiter), which is equivalent to XJupiter. CJupiter is compact in the sense that at a high level, it maintains only a single n-ary digraph that encompasses all replica states.

Much work has been devoted to formal verification of OT functions for lists or trees [9, 10, 23–25]. In contrast, little has been done on the formal verification of complete OT-based protocols. To our knowledge, we are the first to formally specify and verify a family of OT-based Jupiter protocols and the refinement relation among them.

8 Conclusions

We studied a family of OT-based Jupiter protocols for replicated lists. Since OT-based protocols are subtle and error-prone, our work would be helpful to promote a rigorous study of them. We also proposed the AbsJupiter protocol, which addresses the key OT issue in an abstract way. It will be helpful for studying the relation among more OT-based Jupiter protocols.

We will develop a mechanical correctness proof for our AbsJupiter protocol with respect to both strong eventual consistency and weak list specification using

⑪What's different about the new Google Docs: Making collaboration fast. https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html, Sept. 2020.

⑫Attiya H, Gotsman A. Personal communication, 2017. They wrote a note about AJupiter, but have not published it.

TLAPS⑬, a proof system for TLA$^+$. Then we will extend our work to OT-based protocols for replicated lists for P2P systems. In particular, we will study the COT protocol [20] for P2P systems that has inspired us to propose AbsJupiter for client/server systems.

References

[1] Ellis C A, Gibbs S J. Concurrency control in groupware systems. In *Proc. the 1989 ACM SIGMOD International Conference on Management of Data*, May 1989, pp.399-407.

[2] Attiya H, Burckhardt S, Gotsman A, Morrison A, Yang H, Zawirski M. Specification and complexity of collaborative text editing. In *Proc. the 2016 ACM Symposium on Principles of Distributed Computing*, July 2016, pp.259-268.

[3] Nichols D A, Curtis P, Dixon M, Lamping J. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *Proc. the 8th Annual ACM Symposium on User Interface and Software Technology*, November 1995, pp.111-120.

[4] Xu Y, Sun C, Li M. Achieving convergence in operational transformation: Conditions, mechanisms and systems. In *Proc. the 17th ACM Conference on Computer Supported Cooperative Work*, February 2014, pp.505-518.

[5] Shapiro M, Preguiça N, Baquero C, Zawirski M. Conflict-free replicated data types. In *Proc. the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, October 2011, pp.386-400.

[6] Wei H, Huang Y, Lu J. Specification and implementation of replicated list: The Jupiter protocol revisited. In *Proc. the 22nd International Conference on Principles of Distributed Systems*, December 2018, Article No. 12.

[7] Sun C, Ellis C. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proc. the 1998 ACM Conference on Computer Supported Cooperative Work*, November 1998, pp.59-68.

[8] Ressel M, Nitsche-Ruhland D, Gunzenhäuser R. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proc. the 1996 ACM Conference on Computer Supported Cooperative Work*, November 1996, pp.288-297.

[9] Imine A, Rusinowitch M, Oster G, Molli P. Formal design and verification of operational transformation algorithms for copies convergence. *Theor. Comput. Sci.*, 2006, 351(2): 167-183.

[10] Randolph A, Boucheneb H, Imine A, Quintero A. On consistency of operational transformation approach. In *Proc. the 14th International Workshop on Verification of Infinite-State Systems*, August 2012, pp.45-59.

[11] Randolph A, Boucheneb H, Imine A, Quintero A. On synthesizing a consistent operational transformation approach. *IEEE Trans. Computers*, 2015, 64(4): 1074-1089.

[12] Hoare C A. Proof of correctness of data representations. *Acta Inf.*, 1972, 1(4): 271-281.

[13] Lamport L, Merz S. Auxiliary variables in TLA$^+$. arXiv:1703.05121, 2017. https://arxiv.org/pdf/1703.05121.pdf, Sept. 2020.

[14] Lamport L. If you're not writing a program, don't use a programming language. *Bulletin of the EATCS*, 2018, 125: Article No. 7.

[15] Lamport L. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers (1st edition). Addison-Wesley Professional, 2002.

[16] Yu Y, Manolios P, Lamport L. Model checking TLA+ specifications. In *Proc. the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, September 1999, pp.54-66.

[17] Hong W, Chen Z, Yu H, Wang J. Evaluation of model checkers by verifying message passing programs. *SCIENCE CHINA Information Sciences*, 2019, 62(10): Article No. 200101.

[18] Lamport L. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 1994, 16(3): 872-923.

[19] Abadi M, Lamport L. The existence of refinement mappings. *Theor. Comput. Sci.*, 1991, 82(2): 253-284.

[20] Sun D, Sun C. Context-based operational transformation in distributed collaborative editing systems. *IEEE Trans. Parallel Distrib. Syst.*, 2009, 20(10): 1454-1470.

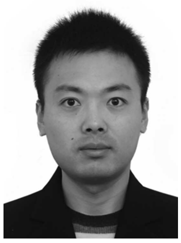
[21] Yuan D, Luo Y, Zhuang X, Rodrigues G R, Zhao X, Zhang Y, Jain P U, Stumm M. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proc. the 11th USENIX Conference on Operating Systems Design and Implementation*, October 2014, pp.249-265.

[22] Li D, Li R. An approach to ensuring consistency in peer-to-peer real-time group editors. *Computer Supported Cooperative Work*, 2008, 17(5/6): 553-611.

[23] Liu Y, Xu Y, Zhang S J, Sun C. Formal verification of operational transformation. In *Proc. the 19th International Symposium on Formal Methods*, May 2014, pp.432-448.

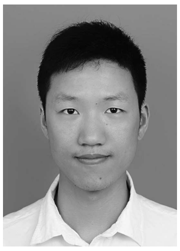
[24] Sun C, Xu Y, Agustina A. Exhaustive search of puzzles in operational transformation. In *Proc. the 17th ACM Conference on Computer Supported Cooperative Work*, February 2014, pp.519-529.

[25] Sinchuk S, Chuprikov P, Solomatov K. Verified operational transformation for trees. In *Proc. the 7th International Conference on Interactive Theorem Proving*, August 2016, pp.358-373.



Heng-Feng Wei received his B.S. and Ph.D. degrees in computer science and technology from Nanjing University, Nanjing, in 2009 and 2016, respectively. He is currently an assistant professor with the Department of Computer Science and Technology and the State Key Laboratory for Novel Software Technology at Nanjing University, Nanjing. His research interests include distributed computing and formal methods. He is a member of CCF.

⑬Microsoft Research — Inria Joint Centre: TLA$^+$ Proof System (TLAPS). https://tla.msr-inria.inria.fr/tlaps/content/Home.html, Sept. 2020.



Rui-Ze Tang received his B.S. degree in computer science and technology from Nanjing University, Nanjing, in 2019. He is currently a Ph.D. candidate with the Department of Computer Science and Technology and the State Key Laboratory for Novel Software Technology at Nanjing University, Nanjing. His research interests include distributed systems and formal methods.

Yu Huang received his B.S. and Ph.D. degrees in computer science from the University of Science and Technology of China, Hefei, in 2002 and 2007, respectively. He is currently a professor with the Department of Computer Science and Technology and the State Key Laboratory for Novel Software Technology at Nanjing University, Nanjing. His research interests include distributed algorithms, distributed systems, formal methods, and system reliability. He is a member of CCF.

Jian Lv received his Ph.D. degree in computer science and technology from Nanjing University, Nanjing. He is currently a professor with the Department of Computer Science and Technology and the director of the State Key Laboratory for Novel Software Technology at Nanjing University, Nanjing. He has served as a vice chairman of the China Computer Federation (CCF) since 2011. His research interests include software methodologies, automated software engineering, and middleware systems. He is a fellow of CCF and a member of ACM.

ISSN 1000-9000(Print)
/1860-4749(Online)
CODEN JCTEEM

JCST

Vol.35 No.6 Nov. 2020

Journal of Computer Science & Technology

SPONSORED BY INSTITUTE OF COMPUTING TECHNOLOGY
THE CHINESE ACADEMY OF SCIENCES &
CHINA COMPUTER FEDERATION
SUPPORTED BY NSFC
CO-PUBLISHED BY SCIENCE PRESS &
SPRINGER

JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY

Volume 35, Number 6, November 2020

Special Section on Software Systems 2020 — Part 2

Regular Paper

JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY
《计算机科学技术学报》

Volume 35 Number 6 2020 (Bimonthly, Started in 1986)

Indexed in: **SCIE, Ei, INSPEC, JST, AJ, MR, CA, DBLP**

Edited by:
THE EDITORIAL BOARD OF JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY
Zhi-Wei Xu, Editor-in-Chief, P.O. Box 2704, Beijing 100190, P.R. China
Managing Editor: Feng-Di Shu E-mail: jcst@ict.ac.cn http://jcst.ict.ac.cn Tel.: 86-10-62610746

Sponsored by: Institute of Computing Technology, CAS & China Computer Federation
Supervised by: Chinese Academy of Sciences
Undertaken by: Institute of Computing Technology, CAS
Published by: Science Press, Beijing, China
Printed by: Beijing Kexin Printing House
Distributed by:
China: All Local Post Offices
Other Countries: Springer Nature Customer Service Center GmbH, Tiergartenstr. 15, 69121 Heidelberg, Germany
Available Online: https://link.springer.com/journal/11390

ISSN 1000-9000

国内统一刊号: CN11-2296/TP 国内邮发代号: 2-578 RMB ￥150.00