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-

⑤Institute of Computing Technology, Chinese Academy of Sciences 2020
ply that clients process operations in the same order. To achieve convergence, Jupiter adopts the operational transformation (OT) technique\textsuperscript{[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\textsuperscript{[1, 3]}.

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\textsuperscript{[1, 8]} were even later shown incorrect\textsuperscript{[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).

- 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.
• 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 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]©. 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 \( \text{Spec} \equiv \text{Init} \land \Box [\text{Next} \land L] \), where \( \text{Init} \) is a predicate specifying all possible initial states of the program, \( \text{Next} \) specifies the next-state relation of the program, \( \Box \) is the temporal operator read “Always”, \( \text{vars} \) 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 \( \text{Next} \) is typically a disjunction of all the actions of the program. The expression \( [\text{Next}]\text{vars} \) is true if \( \text{Next} \) is true, meaning that some action is true and thus taken, or if \( \text{vars} \) stutters, meaning that their values are unchanged. A behavior of the program specified by \( \text{Spec} \) (ignoring \( L \)) of the above form is a sequence of states that satisfy \( \text{Spec} \), namely, the \( \text{Init} \) predicate holds in the first state of this sequence, and the next-state relation \( [\text{Next}]\text{var} \) 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+© 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 \( (F(x_1, \ldots, x_n) \triangleq \cdots) \), and claim theorems (THEOREM). A module \( M \) can import the declarations, definitions, and theorems from other modules \( M_1, \ldots, M_n \) by extending them, namely writing \( \text{EXTENDS } M_1, \ldots, M_n \) in \( M \). Modules can also be instantiated. Let us consider the following instance statement in module \( M \):

\[ IM_1 \triangleq \text{INSTANCE } M_1 \text{ WITH } p_1 \leftarrow e_1, \ldots, p_n \leftarrow e_n, \]

where \( p_1 \) consists of all declared constants and variables of \( M_1 \) and \( e_i \) are valid expressions in \( M_1\). 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

© Note 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].
we can instantiate $\mathbf{Proc}$ by a TLC model value, which is considered to be a finite-state instance of TLA\(^+\) specifications. For example, a TLC model of a specification $\mathbf{AbsSpec}$ that the resulting behavior satisfies $\mathbf{AbsSpec}$\(^+\). 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 $\llbracket y_i \rrbracket$ in terms of $X$ and $Z$, substitute $y_i \leftarrow \llbracket y_i \rrbracket$ in $\mathbf{AbsSpec}$ to get $\mathbf{AbsSpec}$, and we show that $\mathbf{ImplSpec}$ refines $\mathbf{AbsSpec}$. The substitution $y_i \leftarrow \llbracket y_i \rrbracket$ is called a refinement mapping. To verify the assertion that $\mathbf{ImplSpec}$ refines $\mathbf{AbsSpec}$ under such a refinement mapping in TLA\(^+\), we can add the following definition to module $\mathbf{ImplModule}$ ($\mathbf{AbsSub}$ is a fresh identifier).

$$\mathbf{AbsSub} \triangleq \mathbf{instance} \mathbf{AbsModule} \text{ with } y_1 \leftarrow \llbracket y_1 \rrbracket, \ldots, y_n \leftarrow \llbracket y_n \rrbracket.$$  

Then we let TLC check the theorem:

**Theorem** $\mathbf{ImplSpec} \implies \mathbf{AbsSub}!\mathbf{AbsSpec}$,

which is added to module $\mathbf{ImplModule}$.

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\(^{[14]}\).
Constructing a refinement mapping may require adding auxiliary variables to the (lower-level) protocols\cite{13,19}. One kind of auxiliary variables that we will use in data refinement among Jupiter protocols is called history variables\cite{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\cite{13}.

2.2 System Model

We let Client denote the set of client replicas, Server the unique server replica, and Replica $\triangleq\text{Client} \cup \{\text{Server}\}$ 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 $\text{list}[r]$ (initially empty; denoted by $\epsilon$) and interacts with three kinds of actions from users and other replicas.

- **Do($c \in \text{Client}, op \in Op$)**. 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($c \in \text{Client}, m \in M$)**. Client $c$ receives and processes a message $m$ from the server.

- **$S\text{Rev}(m \in M)$**. 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 $\text{Rev}(c_3, \_)$, client $c_3$ receives a message containing the information about $o_1$ (maybe transformed) of client $c_1$ from the server. Next, in $\text{Do}(c_3, o_4)$, it generates operation $o_4$ ($\text{Ins}(h, 2)$), applies $o_4$ locally, and sends $o_4$ to the server. Then, in $\text{Rev}(c_3, \_)$, 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 $\text{list}[c_3]$ at $c_3$ is updated accordingly.

2.3 List, OT, and Weak List Specification

A replicated list object supports two types of update operations: $\text{Del}$ and $\text{Ins}$, defined as records in module $\text{Op}$ (Fig.4). Following \cite{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 $\text{Ins}$ helps to resolve the conflicts caused by two concurrent $\text{Ins}$ operations that are intended to insert different elements at the same position.

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

We consider the weak list specification $\text{WLSpec}\cite{2}$, which is stronger than strong eventual consistency (SEC)\cite{5}. $\text{WLSpec}$ is equivalent to the “pairwise state compatibility property”\cite{8}. 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 $\text{WLSpec}$ (Fig.6) for the formal specification of $\text{Compatible}$). Let $h\text{list}$ be a set of lists. $\text{WLSpec}$ is defined as $\text{WLSpec} \triangleq \forall l_1, l_2 \in h\text{list}: \text{Compatible}(l_1, l_2)$ (see also module $\text{AbsJupiterH}$ in Subsection 6.2).

**Example 3 (Weak List Specification. Adapted from \cite{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 $\text{WLSpec}$, since the lists are pairwise compatible. However, an execution is not allowed by $\text{WLSpec}$, 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**

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

Fig.4. TLA+ module Op.
transform \( op \) against the operations that are concurrent with it and have been previously executed at \( r \) in their serialization order, denoted by \( \text{SO} \), 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 (\( \text{Cop} \)) and context-based OT (\( \text{COT} \)) [20], Jupiter protocols fall into two categories: context-based including \( \text{AbsJupiter}, \ \text{CJupiter}, \ \text{XJupiter} \), and non-context based, i.e., \( \text{AJupiter} \). In this subsection, we define \( \text{Cop} \) and \( \text{COT} \). 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>
<thead>
<tr>
<th>Protocol</th>
<th>Concurrent</th>
<th>SO</th>
<th>Data Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{AbsJupiter} )</td>
<td>( \text{COT} )</td>
<td>SV</td>
<td>Set</td>
</tr>
<tr>
<td>( \text{CJupiter} )</td>
<td>( \text{COT} )</td>
<td>SV</td>
<td>( n )-ary digraph</td>
</tr>
<tr>
<td>( \text{XJupiter} )</td>
<td>( \text{COT} )</td>
<td>COT</td>
<td>2D digraph</td>
</tr>
<tr>
<td>( \text{AJupiter} )</td>
<td>ACK</td>
<td>Buffer</td>
<td>1D buffer</td>
</tr>
</tbody>
</table>

Each operation \( op \in \text{Op} \) is associated with a unique operation identifier (\( \text{oid} \), for short) in \( \text{Oid} \), 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 \( \text{oid} \) whenever the replica \( r \) receives and processes an operation with \( \text{oid} \).
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 $oid_1$ and $oid_2$ that are generated or received by some replica $r$. The operator $so(oid_1, oid_2, sv)$ in module $SV$ (Fig.8) decides whether $oid_1$ precedes (or will precede) $oid_2$ 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 $fceph$ against which $coph$ is to be transformed,
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 \( coph \) is received by some replica, the operations in its context have already been in this replica. Thus, such \( fcoph \) satisfying \( fcoph.ctx = coph.ctx \) 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 \( ctxDiffh \setminus \{ fcoph.oid \} \).

### 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 \( coph \) satisfying \( coph.ctx = e.from \), meaning that when applied, \( coph \) changes the document state from \( e.from \) to \( e.to = e.from \cup \{ coph.oid \} \). The operator \( \oplus \) takes the union of two records with \( node \) and \( edge \) fields.
In CJupiter and XJupiter, when a replica $r$ (either client or server) receives a context-based operation $cop$, it calls $xF orm(NextEdge, r, cop, g)$ 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. $xF ormHelper(uh, vh, coph, gh)$ starts the transformation with $uh \leftarrow u$ (Fig.11 and module $Digraph$).

1) If $uh = ds[r]$, the most recently transformed operation $coph$, the record $gh$ consisting of nodes and edges produced in $xF orm$ 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 $Op$. $xF orm(op, ops)$ of module $Buffer$ (Fig.12) transforms an operation $op$ against a buffer $ops$ of operations (see Fig.13). It utilizes $xF ormOpOps(op, ops)$ and $xF ormOpsOp(ops, op)$ to obtain the last transformed operation $xop$ and the transformed buffer $xops$, respectively. Specifically, $xF ormOpOps$ returns the sequence of intermediate transformed operations, the last one of which is the desired $xop$.

1) If $ops$ is empty, $(op)$ 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 $xF ormOpsOp$ to generate $xops$ by transforming each operation in $ops$ against the corresponding one in $opX \triangleq xFormOpOps(op, ops)$. Finally, $xF ormShift(op, ops, shift)$ 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
Perform\((r, \text{cop})\) calls \textit{xForm}\((r, \text{cop})\) of module \textit{Set} to transform \textit{cop} in \textit{copss}\([r]\). The transformed operation \textit{xform.xcop.op} is applied to \textit{list}[r] and \textit{copss}[r] is updated to \textit{xform.xcopss}.

\hspace{1cm}

![Diagram](image_url)

\textbf{Fig.13.} Illustration of \textit{xForm} of module \textit{Buffer}.

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

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

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

After receiving and applying \(o_3\{\}\) (\textit{INS}(\(\langle x, 1 \rangle\)) of client \(c_1\) from the server, client \(c_3\) generates \(o_3\{\}\) (\textit{INS}(\(b, 2\)). It wraps \(o_4\) into a context-based operation \(o_4\{o_1\}\), adds \(o_4\{o_1\}\) to \textit{copss}[\(c_3\)] = \{\{\}\}, applies \(o_4\) locally, and then sends \(o_4\{o_1\}\) to the server.

Next, client \(c_3\) receives \(o_2\{o_1\}\) (\textit{DEL}(1)) of client \(c_1\) from the server. By \textit{xForm}\((c_3, o_2\{o_1\})\), it transforms \(o_2\{o_1\}\) against the set of context-based operations in \textit{copss}[\(c_3\)] = \{\{\}, \{o_4\{o_1\}\}\}. Since \(o_1\) is the only concurrent operation with \(o_2\) in \textit{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 \(o_2\{o_1, o_2\} (\text{DEL}(1))\) and \(o_4\{o_1, o_2\} (\text{INS}(b, 1))\) are added into \textit{copss}[\(c_3\)]. The transformed operation \textit{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 \textit{xForm}\((c_3, o_3\{o_1\})\), it transforms \(o_3\{o_1\}\) against the set of context-based operations in \textit{copss}[\(c_3\)] = \{\{\}, \{o_4\{o_1\}, o_4\{o_1, o_2\}, o_4\{o_1, o_2, o_3\}\}\}. The set of concurrent operations with \(o_3\) in \textit{copss}[\(c_3\)] is calculated as \(\{o_1, o_2, o_3\} \cup \{o_1\} = \{o_2, o_4\}\). Since \(o_2\) precedes \(o_4\) in the SO order according to \textit{serial}[\(c_3\)] = \{\(o_1, o_2\), \(o_3\{o_1\}\) is first transformed with \(o_2\{o_1\}\), yielding \(o_3\{o_1, o_2\} (\text{INS}(a, 1))\) and \(o_2\{o_1, o_2\}\) (\textit{DEL}(2)). Then, \(o_3\{o_1, o_2\}\) is transformed with \(o_4\{o_1, o_2\}\) (\textit{INS}(b, 1)), yielding \(o_3\{o_1, o_2, o_3\} (\textit{INS}(a, 2))\) and \(o_4\{o_1, o_2, o_3\}\) (\textit{INS}(b, 1)).

At last, \(c_3\) applies the transformed operation \textit{INS}(\(a, 2\)) locally, obtaining the list \(ba\).

4.2 \textit{CJupiter}

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

---

```
module AbsJupiter

VARIABLES copss \textit{copss}[\(r\)]: the set of context-based operations maintained at replica \(r\)

PERFORM\((r, \text{cop})\) \triangleq \text{LET } xform \triangleright xForm\((r, \text{cop})\) \text{ \textit{xform}: } [\textit{xcop}, \textit{xcopss}] \text{ IN } \text{ \textit{copss}'} = \textit{copss} \text{ \textit{EXCEPT} } [\textit{r} = xform.xcopss] \text{ \textit{AND} } \text{apply xform.xcop.op to list}[\textit{r}]

DO\((c, \text{op})\) \triangleq \text{LET } \textit{cop} \triangleright \text{OP} \rightarrow \textit{op}, \textit{oid} \rightarrow \textit{c}, \textit{seq} \rightarrow \textit{csseq}[\textit{c}], \textit{ctx} \rightarrow \textit{ds}[\textit{c}] \text{ IN } \text{ \textit{copss} = } \textit{copss} \text{ \textit{EXCEPT} } [\textit{c} = @ \cup \{\textit{cop}\}] \text{ \textit{AND} } \text{apply op to list}[\textit{c}]; \text{send \textit{cop} to the \textit{Server}}

REV\((c, \text{cop})\) \triangleq \text{PERFORM}(\text{\textit{c}}, \text{\textit{cop}) \text{ \textit{AND} } broadcast \textit{cop} to clients other than ClientOf(\textit{cop})}}
```

---

\textbf{Fig.14.} TLA\(^+\) module \textit{AbsJupiter}.
Except that \( xForm(NextEdge, r, cop, css[r]) \) of module \( \text{Digraph} \) 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

---

**Fig.15. Illustration of client \( c_3 \) in Jupiter protocols under the schedule of Fig.1.** (a) AbsJupiter. (b) CJupiter. (c) XJupiter. (d) AJupiter.
\{o_1, o_2\} by \textit{v}_{14}.

After receiving and applying \textit{a_1} \{\} of client \textit{c_1} redirected by the server, client \textit{c_3} generates \textit{a_4} \{\textit{Ins}(b, 2)\}. It wraps \textit{a_4} into a context-based operation \textit{a_4} \{\textit{a_1}\}, links a new node \textit{v}_{14} to \textit{v}_1 via an edge labeled by \textit{o_1} \{\textit{a_1}\}, and then sends \textit{o_4} \{\textit{a_1}\} to the server.

Next, client \textit{c_3} receives \textit{a_2} \{\textit{a_1}\} \{\textit{Del}(1)\} of client \textit{c_1} from the server. The context of \textit{o_2} \{\textit{a_1}\} matches node \textit{v}_1. By \textit{xForm}, \textit{o_2} \{\textit{a_1}\} and \textit{o_4} \{\textit{a_1}\} are transformed against each other. Node \textit{v}_{124} is created and is linked to \textit{v}_{12} and \textit{v}_{14} via the edges labeled with \textit{o_4} \{\textit{a_1}, \textit{o_2}\} \{\textit{Ins}(b, 1)\} and \textit{o_2} \{\textit{a_1}, \textit{o_4}\} \{\textit{Del}(1)\}, respectively.

Finally, client \textit{c_3} receives \textit{a_3} \{\textit{a_1}\} \{\textit{Ins}(a, 1)\} of client \textit{c_2} from the server. The context of \textit{o_3} \{\textit{a_1}\} matches node \textit{v}_1. By \textit{xForm}, \textit{o_3} \{\textit{a_1}\} will be transformed with the operation sequence consisting of operations along the “first” (in terms of so with \textit{serial}[\textit{c}_3] = \{\textit{o_1}, \textit{o}_2\}) edges from \textit{v}_1 to \textit{v}_{124}. Specifically, \textit{o_3} \{\textit{a_1}\} is first transformed with \textit{o_2} \{\textit{a_1}\}. Then, \textit{o_3} \{\textit{o_1}, \textit{o}_2\} \{\textit{Ins}(a, 1)\} is transformed with \textit{o_4} \{\textit{o_1}, \textit{o}_2\} \{\textit{Ins}(b, 1)\}, yielding \textit{v}_{1234}, \textit{o_3} \{\textit{o_1}, \textit{o}_2, \textit{o}_4\} \{\textit{Ins}(a, 2)\}, and \textit{o_4} \{\textit{o_1}, \textit{o}_2, \textit{o}_3\} \{\textit{Ins}(b, 1)\}. Client \textit{c_3} applies \textit{Ins}(a, 2), obtaining list \textit{ba}.

4.3 XJupiter

XJupiter (Fig. 17) uses 2-dimensional (2D) digraphs where the outdegree of each node is at most 2. Each client \textit{c} maintains a single 2D digraph \textit{c_2ss}[\textit{c}], and the server maintains \textit{n} 2D digraphs, one digraph \textit{ss_2ss}[\textit{c}] per client \textit{c}. Conceptually, a 2D digraph, either \textit{c_2ss}[\textit{c}] or \textit{ss_2ss}[\textit{c}], has two dimensions: a local dimension for storing operations generated by \textit{c} and a remote dimension for storing operations generated by other clients.

In \textit{Do(c, op)}, the client \textit{c} first wraps \textit{op} into a context-based operation \textit{cop} by attaching \textit{oid} and \textit{ctx} = \textit{ds}[\textit{c}] to it. Then it applies \textit{op} to \textit{list}[\textit{c}], inserts an edge labeled by \textit{cop} from node \textit{ds}[\textit{c}] in \textit{ss_2ss}[\textit{c}] along the local dimension, and sends \textit{cop} to the server.

When the server receives a context-based operation \textit{cop} from client \textit{c}, it transforms \textit{cop} against the context-based operations along the remote dimension from node \textit{u} \textit{cop}.ctx to \textit{ds}[\textit{Server}] in \textit{ss_2ss}[\textit{c}]. In \textit{SRev}(cop), this is done in \textit{xForm}(\textit{NextEdge}, \textit{Server}, \textit{cop}, \textit{ss_2ss}[\textit{c}]) of module \textit{Digraph}, where \textit{NextEdge} returns the unique outgoing edge of a given node. Then, the transformed operation \textit{xForm}.\textit{xcop}.\textit{op} is applied to \textit{list}[\textit{Server}], \textit{ss_2ss}[\textit{c}] is updated to integrate \textit{xForm}.\textit{lg}, and \textit{xForm}.\textit{lg} is inserted to the remote dimension of each digraph \textit{ss_2ss}[\textit{c}]. Finally, the server broadcasts the transformed context-based operation \textit{xForm}.\textit{xcop} to other clients except \textit{c}.

When client \textit{c} receives a context-based operation \textit{cop} from the server, it calls \textit{xForm}(\textit{NextEdge}, \textit{c}, \textit{cop}, \textit{ss_2ss}[\textit{c}]) of module \textit{Digraph} to transform \textit{cop} against the operations along the local dimension from node \textit{u} \textit{cop}.ctx to \textit{ds}[\textit{c}] in \textit{c_2ss}[\textit{c}]. The intermediate \textit{xForm}.\textit{lg} is integrated into \textit{c_2ss}[\textit{c}] and the transformed operation \textit{xForm}.\textit{xcop}.\textit{op} is applied to \textit{list}[\textit{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 \cite{[8].}

More importantly, this improvement makes it possible to reduce \textit{n}-ary digraphs to 2D-digraphs.
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 $o_3(o_1) \{\text{INS}(a,1)\}$ to the server. At the server, $o_3(o_1)$ was transformed with $o_2(o_1)$.
(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 \( o_3 \{ a \} \) 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 “\( \rightarrow \)” 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 \( v_{1234}, o_5 \{ o_1, o_2, o_4 \} \) (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 piggy-backed 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 \( AJMsg \)). When a client \( c \) receives a message \( m \) of form \( \{ a \} c \rightarrow c, ack \rightarrow crec[c], op \rightarrow op \) 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 \( \{ c \} \rightarrow c, ack \rightarrow crec[c], op \rightarrow op \) from client \( c \), it knows that among the (transformed) operations in \( cbuf[c] \) 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 \( xFormShift(m.op, cbuf[c], m.ack) \) of module \( Buffer \) to transform \( op \) against the subsequence of operations obtained by shifting the first \( m.ack \) operations out of \( cbuf[c] \). The transformed operation \( xop \) will be appended to other \( cbuf[c] \) for clients \( c \neq c \). Finally, \( Server \) sends the transformed operation \( xop \) along with

![Diagram](image-url)

**Fig.19. TLA+ module \( AJupiter \).**

---

**TABLE: Important Variables and Module Definitions**

<table>
<thead>
<tr>
<th>VARIABLES</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( cbuf, crec, sbuf, src )</td>
<td></td>
</tr>
<tr>
<td>( AJMsg )</td>
<td>[ c : Client, ack : Nat, op : Op \cup { Nop } \cup { } ] from client ( c ) to ( Server )</td>
</tr>
<tr>
<td></td>
<td>[ ack : Nat, op : Op \cup { Nop } ] from ( Server ) to clients</td>
</tr>
</tbody>
</table>

**Module Definitions**

- **\( Do(c, op) \):**
  - \( \land cbuf' = [ cbuf \quad \text{EXCEPT} \quad \{ c \} = \text{Append}(\emptyset, op) ] \)
  - \( \land crec' = [ crec \quad \text{EXCEPT} \quad \{ c \} = 0 ] \)
  - \( \land \text{apply } op \text{ to list}[c] \)

- **\( Rev(c, m) \):**
  - \( \text{LET } xform = xFormShift(m.op, cbuf[c], m.ack) \)
  - \( \land \text{apply } xform \text{ to list}[c] \)

- **\( SRev(m) \):**
  - \( \text{LET } c = m.c \)
  - \( \land zop = xform \text{.zop} \)
  - \( \land \text{apply } zop \text{ to list}[Server] \)
  - \( \land \text{send } \{ ack \rightarrow \text{src}[c], op \rightarrow zop \} \) to client \( c \neq c \)
src[c] to client cl ≠ c.

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

Example 7 (Illustration of AJupiter). We illustrate client c3 in AJupiter under the schedule of Fig. 1 (see also Fig.15(d)).

First, when client c3 receives o1 (INS(x, 1)) of client c1 from the server, its buffer cbuf[c3] is empty. Therefore, in Rec, it simply increases crec[c3] by 1 and applies INS(x, 1) locally.

Next, client c3 generates o4 (INS(b, 2)). In Do, it appends o4 to its currently empty buffer cbuf[c3], resets crec[c3] to 0, applies o4 locally, and sends o4 with ack = 1 to the server.

Then, client c3 receives o2 (DEL(1)) with ack = 0 of client c1 from the server. By xForm of Buffer, o2 (DEL(1)) is transformed against o3 (INS(b, 2)) in buffer cbuf[c3]. The transformed operation OT(o2, o3) = DEL(1) is applied locally, and o4 in buffer cbuf[c3] is transformed into OT(o4, o2) = INS(b, 1).

Finally, client c3 receives transformed o3 (INS(a, 1) which happens to be unchanged) with ack = 0 of client c2 from the server. By xForm of Buffer, o3 (DEL(1)) is transformed against o4 (which is now INS(b, 1)) in buffer cbuf[c3]. The transformed operation OT(o3, o4) = DEL(2) is applied locally, obtaining the list ba. Meanwhile, o4 in buffer cbuf[c3] is transformed into OT(o4, o3) = 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 ∈ Client. Conceptually, this can be expressed in TLA+ as (not syntactically correct):

\[
\text{css[Server]} \leftarrow \text{SetReduce}(\oplus, \text{Range}(s2ss), \text{EmptyGraph}),
\]

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.

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

```plaintext
EXTENDS CJupiter
AbsJ \triangleq INSTANCE AbsJupiter
WITH copss \leftarrow \{ r \in \text{Replica} \rightarrow \{ e.cop : e \in \text{css}[r].edge \} \}
```

---

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.

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

```plaintext
EXTENDS CJupiter
AbsJ \triangleq INSTANCE AbsJupiter
WITH copss \leftarrow \{ r \in \text{Replica} \rightarrow \{ e.cop : e \in \text{css}[r].edge \} \}
```
Heng-Feng Wei et al.: Jupiter Made Abstract, and Then Refined

We have omitted the history variables for recording serial views.

\[ \text{a function mapping an operation (identifier) to the 2D digraph produced during its transformation at the server} \]
\[ \text{cissaX♭c♯: 2D digraph that has been skipped by client c} \]

Fig.22. TLA\[16\] module \( \text{XJupiterImplCJupiter} \).

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

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 \( c\text{issa} \) and \( s\text{issa} \) in \( \text{AJupiterImplXJupiter} \) (Fig.23) which are to simulate \( c\text{issa} \) and \( s\text{issa} \) in XJupiter, respectively. They are supposed to be updated in accordance with \( \text{cbuf} \) and \( \text{sbuf} \) of AJupiter. Specifically, in \( DoImpl(c, op) \), the generated operation \( \text{op} \) is wrapped as a context-based operation \( \text{cop} \) and added to \( c\text{issa} \) as in XJupiter; besides it is stored in \( \text{cbuf} \) 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 \( c\text{issa}[c] \) and \( s\text{issa} \) 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\[1\] Toolbox of version 1.5.7), a model checker for TLA\[1\], 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...
and the number of characters allowed to be inserted. We use the symmetry set for the set \( \text{Char} \) 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 \( \text{AbsJ} \) from \( \text{CJupiter} \) to \( \text{AbsJupiter} \) defined in \( \text{CJupiterImplAbsJupiter} \) by checking that each behavior of \( \text{CJupiter} \) with variables substituted by \( \text{AbsJ} \) is a behavior allowed by \( \text{AbsJupiter} \). The model checking results are shown in Table 3. Similar results on verification of the refinement mappings defined in \( \text{XJupiterImplCJupiter} \) and \( \text{AJupiterImplXJupiter} \) 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 \( \text{AbsJupiter} \) satisfies the weak list specification \( \text{WLSpec} \). To express \( \text{WLSpec} \) in \( \text{TLA}^+ \), we introduce module \( \text{AbsJupiterH} \) (Fig. 24) which extends \( \text{AbsJupiter} \) with a history variable \( \text{hlist} \). \( \text{AbsJupiterH} \) behaves exactly as \( \text{AbsJupiter} \), except that it collects the new list state \( \text{list}' \) in each action into \( \text{hlist} \). We exit TLC when the number of distinct states it examines reaches a threshold \( \theta \). This is supported by a \( \text{TLA}^+ \) Toolbox nightly build as of 01-28-2019 (at 05:56).

---

1. The 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 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.

2. In 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 \( \theta \). This is supported by a \( \text{TLA}^+ \) Toolbox nightly build as of 01-28-2019 (at 05:56).
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

| 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 | θ = 80 000 000 | 05 : 21 : 04 |
| (3, 1) | 17 | 2 785 | 1 288 | 00 : 00 : 00 |
| (3, 2) | 33 | 206 726 218 | 74 737 027 | 05 : 43 : 26 |
| (3, 3) | 18 | 139 943 577 | θ = 80 000 000 | 05 : 18 : 57 |
| (4, 1) | 26 | 194 877 | 61 117 | 00 : 00 : 18 |
| (4, 2) | 21 | 177 451 069 | θ = 80 000 000 | 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 | θ = 80 000 000 | 08 : 01 : 35 |
| (3, 1) | 17 | 2 785 | 1 288 | 00 : 00 : 02 |
| (3, 2) | 33 | 206 726 218 | 74 737 027 | 05 : 50 : 40 |
| (3, 3) | 20 | 139 577 795 | θ = 80 000 000 | 08 : 50 : 52 |
| (4, 1) | 26 | 194 877 | 61 117 | 00 : 00 : 30 |
| (4, 2) | 19 | 175 896 403 | θ = 80 000 000 | 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 | θ = 80 000 000 | 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 | θ = 80 000 000 | 04 : 48 : 23 |
| (4, 1) | 26 | 194 877 | 61 117 | 00 : 00 : 17 |
| (4, 2) | 21 | 176 794 063 | θ = 80 000 000 | 03 : 49 : 58 |
### Table 6. Model Checking Results of Verifying that AbsJupiter Satisfies \( WLSpec \)

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

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 Docs\[^{[11]}\], 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 AJupiter\[^{[12]}\]. 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


\[^{[12]}\) Attiya H, Gotsman A. Personal communication, 2017. They wrote a note about AJupiter, but have not published it.
TLAPS\textsuperscript{5}, a proof system for TLA\textsuperscript{+}. Then we will extend our work to OT-based protocols for replicated lists for P2P systems. In particular, we will study the COT protocol\textsuperscript{20} for P2P systems that has inspired us to propose AbsJupiter for client/server systems.

References

\begin{itemize}
\item [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.
\item [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.
\end{itemize}

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.
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.
Special Section on Software Systems 2020 — Part 2

Preface ................................................. Tao Xie (1231)
ProSy: API-Based Synthesis with Probabilistic Model ................................... Bin-Bin Liu, Wei Dong, Jia-Xin Liu, Yu-Ting Zhang, and Dai-Yan Wang (1234)
Learning Human-Written Commit Messages to Document Code Changes ... Yuan Huang, Nan Jia, Hao-Jie Zhou, Xiang-Ping Chen, Zi-Bin Zheng, and Ming-Dong Tang (1258)
Automatically Identifying Calling-Prone Higher-Order Functions of Scala Programs to Assist Testers ... Yi-Sen Xu, Xiang-Yang Jia, Fan Wu, Lingbo Li, and Ji-Feng Xuan (1278)
Reachability of Pattered Conditional Pushdown Systems ....... Xin Li, Patrick Gardy, Yu-Xin Deng, and Hirohida Seki (1295)
Specification and Verification of the Zab Protocol with TLA+ ............... Jia-Qi Yin, Hai-Biao Zha, and Yuan Fei (1312)
Modelling and Verification of Real-Time Publish and Subscribe Protocol Using UPPAAL and Simulink/Stateflow .......... Qian-Quan Lin, Shu-Ling Wang, Bo-Hua Zhan, and Bin Gu (1324)
Jupiter Made Abstract, and Then Refined ...................... Feng-Feng Wei, Rui-Ze Tang, Yu Huang, and Jian Lv (1343)
Verifying ReLU Neural Networks from a Model Checking Perspective ........................................ Wan-Wei Liu, Fu Song, Tung-Hao-Ran Zhang, and Ji Wang (1365)
Modular Verification of SPARCv8 Code ...................... Jun-Peng Zha, Xin-Yu Feng, and Lei Qiao (1382)
Automatic Buffer Overflow Warning Validation ................................. Feng-Juan Gao, Yu Wang, Lin-Zhang Wang, Zijiang Yang, and Xuan-Dong Li (1406)

Regular Paper

Neural Explainable Recommender Model Based on Attributes and Reviews ........................................... Yv-Yao Liu, Bo Yang, Hong-Bin Pei, and Jing Huang (1446)
Topic Modeling Based Warning Prioritization from Change Sets of Software Repository ................. Jung-Been Lee, Taek Lee, and Hoh Peter In (1461)