Formal Analysis of a Distributed Algorithm for Tracking Progress

Martín Abadi[1,2], Frank McSherry[1],
Derek G. Murray[1], and Thomas L. Rodeheffer[1]

[1] Microsoft Research Silicon Valley
[2] University of California, Santa Cruz

Abstract. Tracking the progress of computations can be both important and delicate in distributed systems. In a recent distributed algorithm for this purpose, each processor maintains a delayed view of the pending work, which is represented in terms of points in virtual time. This paper presents a formal specification of that algorithm in the temporal logic TLA, and describes a mechanically verified correctness proof of its main properties.

1 Introduction

In distributed systems, it is often useful and non-trivial to know how far a computation has progressed. In particular, the problem of termination detection is classic and remains important. More generally, distributed systems often need to detect progress—not just complete termination—for the sake of correctness and efficiency. For example, knowing that a broadcast message has reached all participants in a protocol enables the sender to reclaim memory and other resources associated with the message; similarly, establishing that a certain phase of a computation has completed can contribute to resource management, can inform scheduling decisions, and also enables speculative computation steps to commit and to result in visible output. For such tasks, protocols need to aggregate and share the local views of the system components. Those protocols may operate continuously ("on-the-fly"), or be triggered from time to time by the need to reclaim resources or by external events. In either case, they are often interesting, delicate, crucial for correctness, and worthy of careful design and analysis.

We are presently engaged in research on large-scale, data-parallel distributed computation, and on the development of a system for this purpose, called Naiad [9]. This research explores a declarative dataflow model that supports incremental and iterative computations, with a generalization of the notion of virtual time [5]. According to its original definition, virtual time is

> a global, one-dimensional, temporal coordinate system imposed on a distributed computation; it is used to measure progress and to define synchronization. It may or may not have a connection with real time.

D. Beyer and M. Boreale (Eds.): FMOODS/FORTE 2013, LNCS 7892, pp. 5–19, 2013.

Naiad relaxes this notion by allowing the temporal coordinate system to be based on a partial order, rather than a one-dimensional linear order, as already suggested in passing by Jefferson [5, p407]. Thus, inputs and other pieces of data are associated with time points in this partial order. The use of a partial order avoids unnecessary constraints (false dependencies) that can hinder the progress of computations.

As in prior work on virtual time (e.g., [12]), progress detection is essential to Naiad. Accordingly, the research on Naiad to date includes the design and implementation of a new distributed algorithm for progress detection. This algorithm relies on out-of-band communication (rather than on punctuation within message streams; cf. [2], [13]) for continuously tracking the progress of a computation with respect to partially ordered virtual time (cf. [12]).

For the present purposes, the significance of these characteristics is less important than the fact that we are interested in a new distributed algorithm for progress detection, that a system depends on it, and that we therefore wish to understand it as well as possible. The goal of this paper is to develop a rigorous specification and analysis of the algorithm. The paper presents a formal specification of the algorithm and its properties. Furthermore, it describes a complete, mechanically verified correctness proof for the algorithm. This proof is quite long, as we explain below. So we do not describe all its steps in this paper, but we summarize its main definitions and lemmas, and we briefly describe our approach and experience doing the proof.

We use TLA [7], a well established linear-time temporal logic, and its associated tools [4], [6]. The choice of a linear-time formalism is not in contradiction with the study of a notion of virtual time that relies on a partial order, as this work illustrates. TLA enables a concise and general description of the algorithm. Through its tools, TLA also enables the mechanical verification of our proof.

We believe that this work has a number of benefits.

- As intended, our study has increased confidence in the algorithm, its properties, and its suitability for Naiad.
- This study has also resulted in a detailed, rigorous, and abstract explanation of the algorithm. The generality of the formulation of the algorithm enables us to contemplate other applications, in other systems or even in Naiad as this system evolves. For example, we can identify the essential properties of the algorithm related to the partial order, without making unnecessary assumptions (for example, that the partial order is well-founded, that it is a lattice, or that it is a product of linear orders) which might hold only in particular circumstances.
- Finally, since this study constitutes one of the largest and most difficult applications of TLA to distributed algorithms to date, it has led to new experience with TLA and to detailed feedback to the TLA developers (for instance, on libraries and on performance issues in the TLA proof tools). Some of this feedback has already resulted in improvements to the TLA Toolbox [6].

The next section contains a brief review of TLA. Section 3 describes the algorithm and its main properties. Section 4 presents our proof. Section 5 concludes. A companion technical report contains complete details of the formal specification and proof [11].

2 A Brief Review of TLA

TLA (the Temporal Logic of Actions) [7] combines first-order predicate logic, set theory, and linear-time temporal operators. Figure 1 reviews the TLA notations used in the formulas in this paper.

The TLA Toolbox [6] is an integrated development environment for writing and checking TLA specifications. Specification can include theorems along with their proofs. Proofs are written as sequences of proof steps. The Toolbox includes the TLA Proof System [4], which checks proofs: it mechanically verifies each proof step by constructing proof obligations and discharging them via a number of back-end provers. The Toolbox also includes a standard library of definitions about natural numbers, integers, sequences, and finite sets, along with fundamental theorems about induction over natural numbers.

3 The Algorithm

In this section, we describe our algorithm, first informally and then in TLA. We also state its main safety property.

3.1 Informal Description

Our progress-detection algorithm oversees a computation. Each state of this computation includes a multiset of records, and the computation consists of a sequence of operations that act on this multiset: each operation atomically consumes some of the existing records and replaces them with some output records. The operations and the ordering of the computation as a whole may be non-deterministic. In particular, in a dataflow system such as Naiad, the records contain data, they flow through a graph, and the nodes in the graph asynchronously perform the operations.

In any state of a computation, the existing records may correspond to different stages in the logical progress of the computation. As an example, let us consider a computation that consumes records that contain images, and that processes each image by sequentially applying two functions f_1 and f_2. In an intermediate state of the computation, an input record x that has not yet been processed at all may coexist with the result $f_1(y)$ of applying f_1 to another input record y, and with the result $f_2(f_1(z))$ of applying both f_1 and f_2 to a third input record z. One may think of the records as corresponding to different points in virtual time. These points in virtual time indicate the progress of the computation. In this case, three linearly ordered points will suffice. In general, as in this example,

Comments in TLA appear shaded like this. Declarations of symbols must appear before the symbols are used, and most of the notation resembles that of ordinary mathematics.

Top-level declarations appear at the beginning of a line and can be used subsequently.

CONSTANT *con*		the constant *con*
VARIABLE *var*		the state variable *var*
zerop	$\triangleq$ formula	an operator that takes no arguments
monop(*a*)	$\triangleq$ formula using *a*	an operator that takes one argument
$a \oplus b$	$\triangleq$ formula using *a* and *b*	an infix operator of two arguments

A LET IN formula creates a declaration that can be used within its subformula.

LET *op* $\triangleq$ op-def-formula IN subformula using *op*

TLA uses the ordinary symbols of set theory $\in \notin \cup \cap \{ \}$
and of propositional logic $\wedge \vee \neg \Rightarrow = \neq$ with first-order quantifiers $\forall \exists$
and the common programming syntax IF THEN ELSE
and has standard libraries for natural numbers and integers $+ - < \leq \geq >$ *Nat Int*.

$\wedge$ subformula-1
$\wedge$ subformula-2
$\vdots$
$\wedge$ subformula-n

A conjunction can be written on a series of lines that all begin with $\wedge$ in the same column. A subformula can extend over multiple lines provided it does not intrude on the column used by its leading $\wedge$. The same syntax works for a disjunction with $\vee$.

TLA has syntax for sets and for functions that map one set to another.

SUBSET *A*	the set of all subsets of *A* (powerset)
$\{a \in A : P(a)\}$	the set of all $a \in A$ such that $P(a)$
CHOOSE $a \in A : P(a)$	the arbitrary choice of some $a \in A$ such that $P(a)$
$[A \to B]$	the set of all functions that map *A* to *B*
$[a \in A \mapsto G(a)]$	the function that maps each $a \in A$ to $G(a)$
$[M$ EXCEPT $![d] = F]$	the function the same as *M* except that *d* maps to *F*
$M[a]$	the result of applying *M* to *a*
DOMAIN *M*	the domain of the function *M*

It is possible to declare a recursive function.

LET $M[a \in A] \triangleq$ def using *M* and *a* IN subformula using *M*

A sequence in TLA is a function that maps $1..n$ to the elements of some set, where $n \in Nat$ is the length of the sequence.

$\langle\,\rangle$	the empty sequence
Seq(*D*)	the set of all sequences of *D*
Len(*Q*)	the length of the sequence *Q*
Append(*Q*, *d*)	the result of appending *d* to the sequence *Q*
Tail(*Q*)	the result of removing the first element from the sequence *Q*

TLA has several temporal operators, of which we use two.

$\Box P$	*P* is true now and at all times in the future.
F'	the value of *F* in the next time step

Fig. 1. Brief review of TLA

we assume a set of points of virtual time, with a partial order, and associate each record with a point in virtual time, but the set of points need not be finite, and the partial order need not be linear.

We do require that, if an operation produces a record at one point in virtual time, then the operation has consumed at least one record at a strictly lower

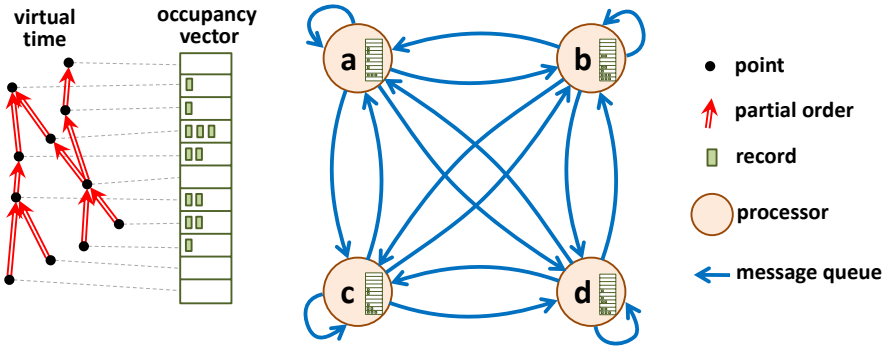

Fig. 2. Overall structure: each processor locally accumulates a delayed view of the occupancy vector

point according to the partial order. Therefore, as a computation proceeds, the population of records will migrate away from lower points. Should a downward-closed set of points become vacant, this set will always thereafter remain vacant, as any operation that might produce a record associated with a point in the set would need to consume such a record as well. (See the safety property *Safe2* in Sect. 4.) This monotonically increasing set of permanently vacant points represents the progress that we wish to track.

We envision that a distributed collection of processors will perform the operations. In such a distributed system, each processor will not be able to observe, directly, the full, exact contents of the set of records in order to measure progress. Processors must instead communicate with one another, as they perform operations, exchanging information about the records that those operations consume and produce. With this information, each processor can maintain a possibly delayed but always safe approximation to the set of permanently vacant points in virtual time.

More concretely, in our algorithm, each processor maintains a local occupancy vector that maps each point to the processor's view of the number of records at that point, depicted in Fig. 2. At the start of a computation, this vector is defined from the initial set of records in the system. A processor tracks the changes in occupancy due to the operations that it performs. When convenient, the processor broadcasts incremental updates to all processors, sending updates about points with net production of records before those about points with net consumption of records. (Section 3.3 describes the exact ordering requirement.) When a processor receives one of these updates, it adjusts its local occupancy vector accordingly. We assume that communication channels between processors are reliable and completely ordered, so that updates are neither dropped nor delivered out of order, and similarly we assume that the processors themselves are reliable; standard communication protocols and fault-tolerance techniques can provide these guarantees.

The intent of this approach is that, once a downward-closed set of points becomes vacant in the local occupancy vector of some processor, that same set

CONSTANT $Point$ set of points
CONSTANT $Proc$ set of processors
CONSTANT $\_ \preceq \_$ partial order on $Point$

$CountVec \triangleq [Point \to Nat]$ count vectors
$DeltaVec \triangleq [Point \to Int]$ delta vectors

$Z \triangleq [t \in Point \mapsto 0]$ everywhere zero
$a \oplus b \triangleq [t \in Point \mapsto a[t] + b[t]]$ component-wise addition
$a \ominus b \triangleq [t \in Point \mapsto a[t] - b[t]]$ component-wise subtraction
$s \prec t \triangleq s \preceq t \wedge s \neq t$ strictly lower

$IsVacantUpto(a,\ t) \triangleq \forall\, s \in Point : s \preceq t \Rightarrow a[s] = 0$
$IsNonposUpto(a,\ t) \triangleq \forall\, s \in Point : s \preceq t \Rightarrow a[s] \leq 0$
$IsSupported(a,\ t) \triangleq \exists\, s \in Point : s \prec t \wedge a[s] < 0 \wedge IsNonposUpto(a,\ s)$
$IsUpright(a) \triangleq \forall\, t \in Point : a[t] > 0 \Rightarrow IsSupported(a,\ t)$

VARIABLE $nrec$ $\in CountVec$
VARIABLE $temp$ $\in [Proc \to DeltaVec]$
VARIABLE msg $\in [Proc \to [Proc \to Seq(DeltaVec)]]$
VARIABLE $glob$ $\in [Proc \to DeltaVec]$

Fig. 3. Basic definitions

of points is in fact vacant thereafter in the global set of records. (We state this safety property formally in Sect. 3.4.) Although the local occupancy vector can be a delayed view of the true occupancy vector, it is a safe approximation, so it allows each processor to report correct results from completed parts of the computation to external observers; it is also a useful input to each processor's memory management and scheduling decisions.

3.2 Basic Definitions

The formal specification of the progress-detection algorithm starts with the basic definitions shown in Fig. 3.

Those definitions introduce three constants: *Point* is the set of points, *Proc* is the set of processors, and $\preceq$ is a partial order on *Point*. There is no requirement that either *Point* or *Proc* be finite.

According to the definitions, a count vector maps each point to a natural number, which represents a count of the number of records at that point. Similarly, a delta vector represents a change in record counts per point. We use Z to designate the delta vector that is everywhere zero and $\oplus$ and $\ominus$ to indicate component-wise addition and subtraction.

We say that a point t in a delta vector a is negative iff $a[t] < 0$, and positive iff $a[t] > 0$. Describing the relative locations of positive and negative points in delta vectors is essential to our proof, so we define several predicates for this purpose. A delta vector a is vacant up t iff $a[s] = 0$ for all $s \preceq t$; it is non-positive up t iff $a[s] \leq 0$ for all $s \preceq t$. A delta vector a is supported at point t iff there exists a negative point $s \prec t$ such that a is non-positive up to s. We then say that s supports t. A delta vector is upright iff all of its positive points are supported.

This definition of upright delta vectors arises because we use delta vectors to describe the changes in record counts that operations cause. As indicated in Sect. 3.1, we require that for any point t at which an operation causes a net production of records there must be a lower point s at which the operation causes a net consumption of records; this property explains why, in an upright delta vector, for each positive point t there must exist a negative point $s \preceq t$. For s to support t, we further require that all points $u \preceq s$ be non-positive; this property prevents cases of infinite descent. It yields, in particular, that the sum of two upright delta vectors is upright. (In cases where $\preceq$ is well-founded, infinite descent is impossible, so the further requirement becomes superfluous.)

Finally, we specify the state of the algorithm using four state variables: *nrec*, *temp*, *msg*, and *glob*.

- *nrec* is the occupancy vector, which represents the number of records that currently exist at each point.
- *temp*[*p*] is the local (temporary) change in the occupancy vector due to the performance of operations at processor p. Note that the change at a given point can be negative (net records consumed), positive (net records produced), or zero. We call it temporary because eventually the processor takes the information from *temp*[*p*] and broadcasts it as an incremental update.
- *msg*[*p*][*q*] is the queue of updates from processor p to processor q. Each update is a delta vector that is zero everywhere except at those points that contain information about net changes. Implementations may of course limit the number of non-zero points and represent updates in a compact form.
- *glob*[*q*] is the delayed view at processor q of the occupancy vector. It is a delta vector, rather than a count vector, because *glob*[*q*][*t*] can be negative for some point t. Such negative values can appear, for example, when one processor p_1 produces a record at point t, a second processor p_2 consumes it and, because of different queuing delays, processor q receives the update from p_2 before that from p_1.

3.3 The Algorithm

Building on the definitions of Fig. 3, Fig. 4 gives the specification of the progress-detection algorithm. It defines an initial condition *Init*, a next-state relation *Next*, and then a complete specification *Spec* which states that *Init* must hold and then forever each step must satisfy the *Next* relation.

Init states that *nrec* can be any mapping from *Point* to *Nat*; this mapping represents an arbitrary initial population of records. Initially, there are no unsent changes, no unreceived updates, and each processor knows the initial population.

Each step from a current state to a next state is an action specified as a relation between the values of the state variables in the current state (unprimed) and in the next state (primed). The algorithm has three actions: *NextPerformOperation*, *NextSendUpdate*, and *NextReceiveUpdate*.

- In the *NextPerformOperation* action, processor p performs an operation that consumes and produces some number of records at each point. The records

$Init \triangleq$
$\quad \wedge\ nrec \in CountVec$ any initial population of records
$\quad \wedge\ temp = [p \in Proc \mapsto Z]$ no unsent changes
$\quad \wedge\ msg = [p \in Proc \mapsto [q \in Proc \mapsto \langle\rangle]]$ no unreceived updates
$\quad \wedge\ glob = [q \in Proc \mapsto nrec]$ each processor knows the initial $nrec$

$NextPerformOperation \triangleq \exists\, p \in Proc,\ c \in CountVec,\ r \in CountVec :$
$\quad$ LET $delta \triangleq r \ominus c$ IN the net change in record population
$\quad\quad \wedge \forall\, t \in Point : c[t] \leq nrec[t]$ only consume what exists
$\quad\quad \wedge\ IsUpright(delta)$ net change must be upright
$\quad\quad \wedge\ nrec' = nrec \oplus delta$
$\quad\quad \wedge\ temp' = [temp$ EXCEPT $![p] = temp[p] \oplus delta]$
$\quad\quad \wedge$ UNCHANGED msg
$\quad\quad \wedge$ UNCHANGED $glob$

$NextSendUpdate \triangleq \exists\, p \in Proc,\ tt \in$ SUBSET $Point :$
$\quad$ LET $gamma \triangleq [t \in Point \mapsto$ IF $t \in tt$ THEN $temp[p][t]$ ELSE $0]$ IN
$\quad\quad \wedge\ gamma \neq Z$ update must say something
$\quad\quad \wedge\ IsUpright(temp[p] \ominus gamma)$ what is left must be upright
$\quad\quad \wedge$ UNCHANGED $nrec$
$\quad\quad \wedge\ temp' = [temp$ EXCEPT $![p] = temp[p] \ominus gamma]$
$\quad\quad \wedge\ msg' = [msg$ EXCEPT $![p] = [q \in Proc \mapsto Append(msg[p][q],\ gamma)]]$
$\quad\quad \wedge$ UNCHANGED $glob$

$NextReceiveUpdate \triangleq \exists\, p \in Proc,\ q \in Proc :$
$\quad$ LET $kappa \triangleq msg[p][q][1]$ IN oldest unreceived update from p to q
$\quad\quad \wedge\ msg[p][q] \neq \langle\rangle$ message queue must be non-empty
$\quad\quad \wedge$ UNCHANGED $nrec$
$\quad\quad \wedge$ UNCHANGED $temp$
$\quad\quad \wedge\ msg' = [msg$ EXCEPT $![p][q] = Tail(msg[p][q])]$
$\quad\quad \wedge\ glob' = [glob$ EXCEPT $![q] = glob[q] \oplus kappa]$

$Next \triangleq NextPerformOperation \vee NextSendUpdate \vee NextReceiveUpdate$

$Spec \triangleq Init \wedge \Box Next$

Fig. 4. Formal specification of the progress-detection algorithm

to be consumed must exist and the net change in records *delta* must be an upright delta vector. The action adds *delta* to *nrec* and to $temp[p]$.

- In the *NextSendUpdate* action, processor p selects a set of points *tt* and broadcasts an update about its changes at those points. The update is represented by *gamma*. The processor must choose *tt* in such a way that $temp[p] \ominus gamma$ is upright. This requirement holds, in particular, when *tt* consists of positive points in $temp[p]$ if any exist, because $temp[p]$ is always upright, as we show in Sect. 4. The action subtracts *gamma* from $temp[p]$ and appends *gamma* to $msg[p][q]$ for all q.
- In the *NextReceiveUpdate* action, processor q selects a processor p and receives the oldest update *kappa* on the message queue from p to q. For this action to take place, the current message queue $msg[p][q]$ must be non-empty. The action adds *kappa* to $glob[q]$ and removes it from $msg[p][q]$.

For any point t and processor q, if $glob[q]$ is vacant up to t, then, at this and all future times, *nrec* is vacant up to t.

$Safe \triangleq \forall\, t \in Point,\ q \in Proc : (\ IsVacantUpto(glob[q], t) \Rightarrow \Box IsVacantUpto(nrec, t)\ )$

An execution that obeys *Spec* is always *Safe*.

THEOREM $Spec \Rightarrow \Box Safe$

Fig. 5. Main safety property of the progress-detection algorithm

The next-state relation *Next* is simply the disjunction of the relations for these three actions.

An implementation may refine this specification in many ways. In particular, while each of the three actions in the specification is atomic, an implementation may perform smaller steps. An implementation may also exhibit less nondeterminism, for example by restricting the choice of *tt* in *NextSendUpdate*. As usual, our results automatically carry over to all correct implementations of the specification.

3.4 The Main Safety Property

The main goal of the progress-detection algorithm is to enable each processor q to deduce global information about the present and future of *nrec* from current, local information about $glob[q]$, as explained in Sect. 3.1. Specifically, if $glob[q]$ is vacant up to t then *nrec* should also be vacant up to t at this and all future times. The main correctness property of the algorithm is that this implication always holds. Figure 5 expresses the implication as a TLA formula *Safe*. Our main theorem is that $\Box Safe$ holds in every execution that obeys *Spec*.

In addition to this safety property, the algorithm satisfies a liveness property: if progress happens then it will eventually be known to all processors, under suitable fairness assumptions on the transmission of updates and other actions. We regard this liveness property as important but less crucial than the safety property, and its treatment would require additional notations, definitions, and arguments, so we do not consider it further in this paper.

4 Formal Verification (Summary)

In this section, we present our formal proof of the correctness of the progress-detection algorithm, that is, of the theorem $Spec \Rightarrow \Box Safe$ stated in Fig. 5. As indicated in the Introduction, the proof is quite long, so we summarize it, describing its argument and stating key auxiliary safety properties. We also comment on various characteristics of the proof and on our experience.

4.1 Proof Summary

As usual, invariants are predicates on states that always hold in every execution that obeys the specification, and we talk about "proving an invariant" when

For any point t, if *nrec* is vacant up to t, then it will be so at all future times.
$Safe2 \triangleq \forall\, t \in Point : (\, IsVacantUpto(nrec, t) \Rightarrow \Box IsVacantUpto(nrec, t)\,)$

For any point t and processor q, if $glob[q]$ is vacant up to t, then so is *nrec*.
$Inv1 \triangleq \forall\, t \in Point,\ q \in Proc : (\, IsVacantUpto(glob[q], t) \Rightarrow IsVacantUpto(nrec, t)\,)$

Fig. 6. Auxiliary safety property and invariant

what we mean is proving that the state predicate in question is in fact an invariant. To prove an invariant I, it suffices to show that the invariant holds in the initial state ($Init \Rightarrow I$) and that every step that satisfies the next-state relation maintains the invariant ($I \wedge Next \Rightarrow I'$). It follows by induction that I is true in every reachable state. Proving a general safety property is more complicated, but similar deduction rules apply.

The safety property *Safe* follows from an auxiliary safety property *Safe2* and an invariant *Inv1* defined in Fig. 6. *Safe2* says that, whenever *nrec* is vacant up to some point t, it will stay that way. This safety property is a simple consequence of the two requirements $\forall\, t \in Point : c[t] \leq nrec[t]$ and $IsUpright(delta)$ in *NextPerformOperation*, which is the only action that changes *nrec*. *Inv1* says that, for any q, if $glob[q]$ is vacant up to some point t, then so is *nrec*. We devote the rest of this summary to explaining the proof of *Inv1*, which is much harder than that of *Safe2*.

In order to prove *Inv1*, we consider the relation between *nrec*, $glob[q]$, and all of the information about changes to *nrec* that has not yet been incorporated into $glob[q]$. For this purpose, we define $Info(k, p, q)$ as the suffix of information from processor p heading toward processor q that skips the k oldest unreceived updates, and $AllInfo(q)$ as the sum of all information heading toward processor q. Figure 7 introduces the necessary definitions and a lemma, and Fig. 8 asserts several additional auxiliary invariants. Next we discuss the lemma, the auxiliary invariants, and the derivation of *Inv1*, summarizing informally the reasoning that our proof makes formally:

- The lemma states that the sum of two upright delta vectors a and b is upright. This lemma is proved by a case analysis on the location of positive and negative points in the sum. A positive point t in $a \oplus b$ can occur only where at least one of a or b is positive. Without loss of generality, let $a[t] > 0$. Then, since a is upright, there must be a point s that supports t in a. It follows that $a[s] < 0$ and a is non-positive up to s. If b is non-positive up to s then s supports t in $a \oplus b$. Otherwise, there is some $u \preceq s$ such that $b[u] > 0$. Since b is upright, there must be a point v that supports u in b. We can conclude that $(a \oplus b)[v] < 0$ and $a \oplus b$ is non-positive up to v, so v supports t in $a \oplus b$. In either case, there exists a point that supports t in $a \oplus b$. Therefore, $a \oplus b$ is upright.
- *Inv2* states that the sum of $glob[q]$ plus all information heading toward q is *nrec*. This predicate is trivially true in the initial state. *NextPerformOperation* transfers *delta* from *nrec* to $temp[p]$; *NextSendUpdate* transfers

Sum of the sequence Q of delta vectors, skipping the first k. This operator is defined for all $k \in Nat$; the result is Z when $k \geq Len(Q)$. A recursive function computes the sum.

$$
\begin{array}{l}
SumSeq(k,\, Q) \triangleq \\
\quad \text{LET } Elem(i) \triangleq \text{IF } k < i \land i \leq Len(Q) \text{ THEN } Q[i] \text{ ELSE } Z \text{ IN} \\
\quad \text{LET } Sumv[i \in Nat] \triangleq \text{IF } i = 0 \text{ THEN } Z \text{ ELSE } Sumv[i-1] \oplus Elem(i) \text{ IN} \\
\quad Sumv[Len(Q)]
\end{array}
$$

Given a finite set choose a sequence in which each element appears exactly once.

$$
\begin{array}{ll}
ExactSeqFor(D) \triangleq \text{CHOOSE } Q \in Seq(D) : & \\
\quad \land \forall\, d \in D : \exists\, i \in \text{DOMAIN } Q : Q[i] = d & \text{each element appears} \\
\quad \land \forall\, i, j \in \text{DOMAIN } Q : Q[i] = Q[j] \Rightarrow i = j & \text{there are no duplicates}
\end{array}
$$

Sum of the delta vectors in the range of F, assuming only finitely many are non-zero.

$$
\begin{array}{l}
SumFun(F) \triangleq \text{LET } I \triangleq ExactSeqFor(\{d \in \text{DOMAIN } F : F[d] \neq Z\}) \text{ IN} \\
\quad SumSeq(0,\, [i \in \text{DOMAIN } I \mapsto F[I[i]]])
\end{array}
$$

The suffix of information from p heading toward q that skips the k oldest unreceived updates. The sum includes $temp[p]$, which is information that p knows but has not yet sent.

$$Info(k,\, p,\, q) \triangleq SumSeq(k,\, msg[p][q]) \oplus temp[p]$$

All information heading toward q.

$$AllInfo(q) \triangleq SumFun([p \in Proc \mapsto Info(0,\, p,\, q)])$$

The sum of upright delta vectors is upright.

$$\text{LEMMA } \forall\, a,\, b \in DeltaVec : IsUpright(a) \land IsUpright(b) \Rightarrow IsUpright(a \oplus b)$$

Fig. 7. Additional definitions and lemma

For any processor q, the sum of $glob[q]$ plus all information heading toward q is $nrec$.

$$Inv2 \triangleq \forall\, q \in Proc : nrec = glob[q] \oplus AllInfo(q)$$

For any processor p, $temp[p]$ is upright.

$$Inv3 \triangleq \forall\, p \in Proc : IsUpright(temp[p])$$

For any processors p and q, any suffix of information from p heading toward q is upright.

$$Inv4 \triangleq \forall\, k \in Nat,\, p \in Proc,\, q \in Proc : IsUpright(Info(k,\, p,\, q))$$

For any processor q, non-zero information is coming from only a finite set of processors.

$$Inv5 \triangleq \forall\, q \in Proc : IsFiniteSet(\{p \in Proc : Info(0, p, q) \neq Z\})$$

For any processor q, all information heading toward q is upright.

$$Inv6 \triangleq \forall\, q \in Proc : IsUpright(AllInfo(q))$$

For any point t, $nrec[t] \geq 0$.

$$Inv7 \triangleq \forall\, t \in Point :\ nrec[t] \geq 0$$

Fig. 8. Additional auxiliary invariants

For any point t, if $glob[q]$ is vacant up to t, then it will be so at all future times.

$$Safe3 \triangleq \forall\, q \in Proc,\, t \in Point :\ (\ IsVacantUpto(glob[q], t) \Rightarrow \Box IsVacantUpto(glob[q], t)\ )$$

Fig. 9. An extra safety property proved in the full proof

gamma from $temp[p]$ to $msg[p][q]$; and *NextReceiveUpdate* transfers *kappa* from $msg[p][q]$ to $glob[q]$. Each of these actions maintains *Inv2*.

- *Inv3* states that $temp[p]$ is upright. This predicate is true in the initial state because $IsUpright(Z)$ holds. *NextPerformOperation* adds an upright delta vector to $temp[p]$; *NextSendUpdate* requires $IsUpright(temp[p]')$; and *NextReceiveUpdate* has no effect on $temp[p]$.
- *Inv4* states that $Info(k, p, q)$ is upright. This predicate is trivially true in the initial state. *NextPerformOperation* increases $Info(k, p, q)$ by an upright delta vector. *NextReceiveUpdate* shifts everything one position forward in message queue $msg[p][q]$, resulting in $Info(k, p, q)' = Info(k + 1, p, q)$. Both of these actions maintain *Inv4*. The only complicated case is that of *NextSendUpdate*, which transfers *gamma* from $temp[p]$ to the end of $msg[p][q]$. For $k \leq Len(msg[p][q])$, this action makes no change in $Info(k, p, q)$. For $k > Len(msg[p][q])$, it results in $Info(k, p, q)' = temp[p]'$, so it maintains *Inv4* by *Inv3*.
- *Inv5* states that the set of processors with non-zero information heading toward processor q is finite. This predicate is trivially true in the initial state and is clearly maintained by each action.
- *Inv6* states that $AllInfo(q)$ is upright. This invariant follows easily from *Inv4* and *Inv5*.
- *Inv7* states that $nrec[t] \geq 0$ for all t. This predicate is trivially true in the initial state and is maintained by the requirement $c[t] \leq nrec[t]$ in *NextPerformOperation*, which is the only action that changes *nrec*.

Finally, we derive *Inv1* from *Inv2*, *Inv6*, and *Inv7* via a proof by contradiction, as follows. Given a point t such that $glob[q]$ is vacant up to t, suppose that there was a point $s \preceq t$ such that $nrec[s] \neq 0$. By *Inv7*, we would have $nrec[s] > 0$. By *Inv2*, we would have $AllInfo(q)[s] > 0$. By *Inv6*, $AllInfo(q)$ is upright, so there would be a point $u \preceq s$ such that $AllInfo(q)[u] < 0$. But then, by *Inv2*, we would have $nrec[u] < 0$, which is impossible by *Inv7*. Hence there can be no such point s.

4.2 Discussion

Our mechanically verified formal proof contains many more steps and much more detail than the summary presented above. We also went on to prove an extra safety property *Safe3* (shown in Fig. 9), which states that whenever $glob[q]$ is vacant up to t it stays that way. The proof of *Safe3* requires its own additional set of supporting definitions and auxiliary properties.

The entire proof is quite long, requiring 208 pages. There are several reasons for this length.

- We did not try to shorten the proof, e.g., by factoring out lemmas afterwards.
- We often had to decompose proof steps, manually, into several smaller steps that could be verified by one of the proof system's back-end provers.
- The proof contains material that logically should belong in general libraries.

Table 1. Statistics of the formal proof

	modules	theorems	lines	run time (sec)	obligations proved by isabelle	smt3	zenon
library	8	75	4848	1945	38	196	1391
specific	19	71	5895	2450	20	86	1258
Total	**27**	**146**	**10743**	**4395**	**58**	**282**	**2649**

The TLA proof system consists of a proof manager, which parses the TLA files, constructs a set of proof obligations for each proof step, and then calls on various back-end provers to verify each proof obligation. By default, the proof manager first invokes Zenon [1], a tableau prover for classical first-order logic with equality. Zenon is generally quick to solve simple problems, but tends to fail on anything complicated. If Zenon fails, the proof manager then invokes Isabelle [10] using a specialized TLA object logic that includes propositional and first-order logic, elementary set theory, functions, and the construction of natural numbers. Pragmas can be used to direct the proof manager to appeal to other back-end provers. An entire category of provers based on Satisfiability Modulo Theory (SMT) is especially good at some hard problems that involve arithmetic, uninterpreted functions, and quantifiers. The back-end prover smt3 uses one such SMT prover [8].

In an early phase of our research, Leslie Lamport experimented with the use of SMT provers for establishing properties of delta vectors. He obtained some elegant, succinct proofs, which encouraged our effort. As our research progressed, we generalized the specification and weakened the hypotheses; our arguments became longer. In all, we spent about four man-months writing, revising, and debugging the entire formal proof.

To enable the proof manager to deal with the total number of proof obligations, we divided the proof into separate modules. Table 1 gives some statistics. We classify the modules into those that logically could be considered as general library modules and those whose applicability is limited to this specific proof. The entire proof can be checked in less than two hours on an 2.67 GHz Intel® Core™ i7 with 4 GB of RAM. Zenon checks almost 90% of the proof obligations; this fact confirms the effectiveness of Zenon. Just over half of the proof obligations appear in modules that could be considered as library modules.

It was particularly challenging to construct and to study the TLA definitions of what it means to sum up the suffix of a sequence of delta vectors and of what it means to sum up the delta vectors in the range of a function. For this purpose, we had to introduce many theorems with formal proofs about these sums and how they are affected by changes in the underlying sequence or function. For example, given $k \in Nat$, $Q \in Seq(DeltaVec)$, $k \leq Len(Q)$, and $a \in DeltaVec$, we have

$$SumSeq(k, Append(Q, a)) = SumSeq(k, Q) \oplus a$$

Establishing such properties was quite tedious. To a human, on the other hand, these properties are obvious consequences of the fact that $\oplus$ is a commutative

and associative operator with an identity and is closed over *DeltaVec*. In other words, $(DeltaVec, \oplus)$ is a commutative monoid.

In fact, the definitions of *SumSeq* and *SumFun* and related theorems could be written in a general form as a library with application to any commutative monoid. We attempted to construct such a library, but soon discovered a performance bug in the proof manager. The problem was that our desired library of theorems created a collection of TLA modules with an exponential explosion of paths through nested module dependencies. The proof manager ended up spending an enormous amount of time exploring these paths to match proof obligations against known facts.

We presented an example of such behavior to the implementors of the proof manager. Eventually, Damien Doligez solved the performance problem, improving the TLA proof system so that it could handle the library structure we had desired. However, by then, we had completed our proof using specialized *SumSeq* and *SumFun* theorems in a linear chain of modules.

Another, more superficial limitation in the current TLA proof system is that it does not directly capture temporal reasoning. Specifically, for an invariant I, the proof system can check proofs of $Init \Rightarrow I$ and $I \wedge Next \Rightarrow I'$, but the trivial, routine step from these formulas to $Spec \Rightarrow \Box I$ remains manual.

Despite such difficulties, constructing a formal proof with the support of the TLA tools enabled us to examine closely the details of the progress-detection algorithm and to refine them. In particular, in our first attempt at the proof we relied on a well-founded partial order. As our research advanced, we realized that we could remove the requirement of well-foundedness from the partial order and instead introduce the more liberal concept of uprightness for delta vectors. Similarly, we originally had restrictive requirements on updates and on their ordering. In the course of the proof, we realized that the essential requirement is that $temp[p]$ must always be upright. Such refinements enable a broad range of useful implementations.

5 Conclusion

Formal specifications and correctness proofs are often beneficial in the study of distributed algorithms; the work presented in this paper is not an exception in this respect. Among the many other examples in this area, some pertain to tasks related to progress detection. For instance, Chandy and Misra developed a concise, manual correctness argument for an algorithm for termination detection, in Unity [3]. However, we are not aware of any work directly analogous to ours. In particular, the early research on the Time Warp mechanism includes a correctness argument for an algorithm for estimating a global virtual time (with a linear order) [12]; that argument was relatively brief, not fully formal, and manual.

Therefore, beyond its intrinsic results, we regard our analysis as an informative datapoint on the use of formal specifications and proof tools. Although we might wish that our work had been easier, we did complete it with reasonable effort. Ongoing improvements in tools should simplify similar work in the future.

Acknowledgments. We are grateful to Paul Barham, Damien Doligez, Rebecca Isaacs, Michael Isard, Leslie Lamport, and Stephan Merz for discussions about Naiad, the progress tracking protocol, and the TLA proof system.

References

1. Bonichon, R., Delahaye, D., Doligez, D.: Zenon: An extensible automated theorem prover producing checkable proofs. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 151–165. Springer, Heidelberg (2007)
2. Chandramouli, B., Goldstein, J., Maier, D.: On-the-fly progress detection in iterative stream queries. Proc. VLDB Endow. 2(1), 241–252 (2009)
3. Chandy, K.M., Misra, J.: Proofs of distributed algorithms: An exercise. In: Hoare, C.A.R. (ed.) Developments in Concurrency and Communication, pp. 305–332. Addison-Wesley, Boston (1990)
4. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: Verifying safety properties with the TLA+ proof system. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 142–148. Springer, Heidelberg (2010)
5. Jefferson, D.R.: Virtual time. ACM Trans. Program. Lang. Syst. 7(3), 404–425 (1985)
6. Lamport, L.: The TLA Toolbox, `http://research.microsoft.com/en-us/um/people/lamport/tla/toolbox.html`
7. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2002)
8. Merz, S., Vanzetto, H.: Automatic verification of TLA+ proof obligations with SMT solvers. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 289–303. Springer, Heidelberg (2012)
9. Naiad: Web page, `http://research.microsoft.com/en-us/projects/naiad/`
10. Paulson, L.C.: Isabelle: A Generic Theorem Prover. LNCS, vol. 828. Springer, Heidelberg (1994)
11. Rodeheffer, T.L.: The Naiad clock protocol: Specification, model checking, and correctness proof. Tech. Rep. MSR-TR-2013-20, Microsoft Research, Redmond (February 2013), `http://research.microsoft.com/apps/pubs/?id=183826`
12. Samadi, B.: Distributed Simulation, Algorithms and Performancs Analysis. Ph.D. thesis, University of California, Los Angeles (1985), Tech. Rep. CSD-850006, `http://ftp.cs.ucla.edu/tech-report/198_-reports/850006.pdf`
13. Tucker, P.A., Maier, D., Sheard, T., Fegaras, L.: Exploiting punctuation semantics in continuous data streams. IEEE Trans. Knowl. Data Eng. 15(3), 555–568 (2003)