Verified Progress Tracking for Timely Dataflow (Extended Report)

Matthias Brun
Department of Computer Science, ETH Zürich, Switzerland

Sára Decova
Department of Computer Science, ETH Zürich, Switzerland

Andrea Lattuada
Department of Computer Science, ETH Zürich, Switzerland

Dmitriy Traytel
Department of Computer Science, University of Copenhagen, Denmark

Abstract

Large-scale stream processing systems often follow the dataflow paradigm, which enforces a program structure that exposes a high degree of parallelism. The Timely Dataflow distributed system supports expressive cyclic dataflows for which it offers low-latency data- and pipeline-parallel stream processing. To achieve high expressiveness and performance, Timely Dataflow uses an intricate distributed protocol for tracking the computation's progress. We modeled the progress tracking protocol as a combination of two independent transition systems in the Isabelle/HOL proof assistant. We specified and verified the safety of the two components and of the combined protocol. To this end, we identified abstract assumptions on dataflow programs that are sufficient for safety and were not previously formalized.

2012 ACM Subject Classification Security and privacy → Logic and verification; Computing methodologies → Distributed algorithms; Software and its engineering → Data flow languages

Keywords and phrases safety, distributed systems, timely dataflow, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.18

Supplementary Material `https://www.isa-afp.org/entries/Progress_Tracking.html`

1 Introduction

The dataflow programming model represents a program as a directed graph of interconnected operators that perform per-tuple data transformations. A message (an incoming datum) arrives at an input (a root of the dataflow) and flows along the graph's edges into operators. Each operator takes the message, processes it, and emits any resulting derived messages.

This model enables automatic and seamless parallelization of tasks on large multiprocessor systems and cluster-scale deployments. Many research-oriented and industry-grade systems have employed this model to describe a variety of large scale data analytics and processing tasks. Dataflow programming models with timestamp-based, fine-grained coordination, also called time-aware dataflow [23], incur significantly less intrinsic overhead [25].

In a time-aware dataflow system, all messages are associated with a timestamp, and operator instances need to know up-to-date (timestamp) *frontiers*—lower bounds on what timestamps may still appear as their inputs. When informed that all data for a range of timestamps has been delivered, an operator instance can complete the computation on input data for that range of timestamps, produce the resulting output, and retire those timestamps.

A *progress tracking mechanism* is a core component of the dataflow system. It receives information on outstanding timestamps from operator instances, exchanges this information with other system workers (cores, nodes) and disseminates up-to-date approximations of the frontiers to all operator instances.

The progress tracking mechanism must be correct. Incorrect approximations of the frontiers can result in subtle concurrency errors, which may only appear under certain load and deployment circumstances. In this work, we formally model in Isabelle/HOL and prove the safety of the progress tracking protocol of *Timely Dataflow* [1, 25, 26] (Section 2), a time-aware dataflow programming model and a state-of-the-art streaming, data-parallel, distributed data processor.

In Timely Dataflow's progress tracking, worker-local and distributed coordination are intertwined, and the formal model must account for this asymmetry. Individual agents (operator instances) on a worker generate coordination updates that have to be asynchronously exchanged with all other workers, and then propagated locally on the dataflow structure to provide local coordination information to all other operator instances.

This is an additional (worker-local) dimension in the specification when compared to well-known distributed coordination protocols, such as Paxos [20] and Raft [27], which focus on the interaction between symmetric communicating parties on different nodes. In contrast our environment model can be simpler, as progress tracking is designed to handle but not recover from fail-stop failures or unbounded pauses: upon crashes, unbounded stalls, or reset of a channel, the system stops without violating safety. [1]

Abadi et al. [4] formalize and prove safety of the distributed exchange component of progress tracking in the TLA$^+$ Proof System. We present their *clocks protocol* through the lens of our Isabelle re-formalization (Section 3) and show that it subtly fails to capture behaviors supported by Timely Dataflow [25, 26]. We then significantly extend the formalized protocol (Section 4) to faithfully model Timely Dataflow's modern reference implementation [1].

The above distributed protocol does not model the dataflow graph, operators, and timestamps within a worker. Thus, on its own it is insufficient to ensure that up-to-date frontiers are delivered to all operator instances. To this end, we formalize and prove the safety of the *local propagation* component (Section 5) of progress tracking, which computes and updates frontiers for all operator instances. Local propagation happens on a single worker, but operator instances act as independent asynchronous actors. For this reason, we also employ a state machine model for this component. Along the way, we identify sufficient criteria on dataflow graphs, that were previously not explicitly (or only partially) formulated for Timely Dataflow.

Finally, we combine the distributed component with local propagation (Section 6) and formalize the global safety property that connects initial timestamps to their effect on the operator frontier. Specifically, we prove that our combined protocol ensures that frontiers always constitute safe lower bounds on what timestamps may still appear on the operator inputs.

Related Work

Data management systems verification Timely Dataflow is a system that supports low-latency, high-throughput data-processing applications. Higher level libraries [23, 24] and SQL abstractions [2] built on Timely Dataflow support high performance incremental view maintenance for complex queries over large datasets. Verification and formal methods efforts in the data management and processing space have focused on SQL and query-language semantics [6, 10, 12] and on query runtimes in database management systems [7, 22].

Distributed systems verification Timely Dataflow is a distributed, concurrent system: our modeling and proof techniques are based on the widely accepted state machine model and refinement approach as used, e.g., in the TLA$^+$ Proof System [9] and Ironfleet [15]. Recent

[1] Systems based on Timely Dataflow and progress tracking can recover by re-starting the protocol.

work focuses on proving consistency and safety properties of distributed storage systems [13, 14, 21] and providing tools for the implementation and verification of general distributed protocols [19, 30] leveraging domain-specific languages [29, 32] and advanced type systems [16].

Model of Timely Dataflow Abadi and Isard [3] define abstractly the semantics of a Timely Dataflow programming model [25]. Our work is complementary; we concretely compute their *could-result-in* relation (Section 6) and formally model the implementation's core component.

2 Timely Dataflow and Progress Tracking

Our formal model follows the progress tracking protocol of the modern Rust implementation of Timely Dataflow [1]. The protocol has evolved from the one reported as part of the classic implementation Naiad [25]. Here, we provide an informal overview of the basic notions, for the purpose of supporting the presentation of our formal model and proofs.

Dataflow graph A Timely Dataflow computation is represented by a graph of operators, connected by channels. Each worker in the system runs an instance of the entire dataflow graph. Each instance of an operator is responsible for a subset, or shard, of the data being processed. Workers run independently and only communicate through reliable message queues—they act as communicating sequential processes [17]. Each worker alternately executes the progress tracking protocol and the operator's processing logic. Figure 1 shows a Timely Dataflow operator and the related concepts described in this section.

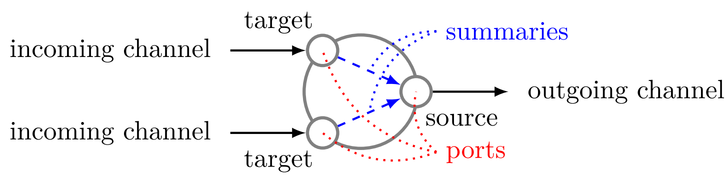

Figure 1 A Timely Dataflow operator.

Pointstamps A pointstamp represents a datum at rest at an operator, or in motion on one of the channels. A pointstamp (l, t) refers to a location l in the dataflow and a timestamp t. Timestamps encode a semantic (causal) grouping of the data. For example, all data resulting from a single transaction can be associated with the same timestamp. Timestamps are usually tuples of positive integers, but can be of any type for which a partial order $\preceq$ is defined.

Locations and summaries Each operator has an arbitrary number of input and output ports, which are locations. An operator instance receives new data through its input ports, or target locations, performs processing, and produces data through its output ports, or source locations. A dataflow channel is an edge from a source to a target. Internal operator connections are edges from a target to a source, which are additionally described by one or more summaries: the minimal increment to timestamps applied to data processed by the operator.

Frontiers Operator instances must be informed of which timestamps they may still receive from their incoming channels, to determine when they have a complete view of data associated with a certain timestamp. The progress tracking protocol tracks the system's pointstamps and summarizes them to one frontier per operator port. A frontier is a lower bound on the timestamps that may appear at the operator instance inputs. It is represented by an antichain F indicating that the operator may still receive any timestamp t for which $\exists t' \in F.\ t' \preceq t$.

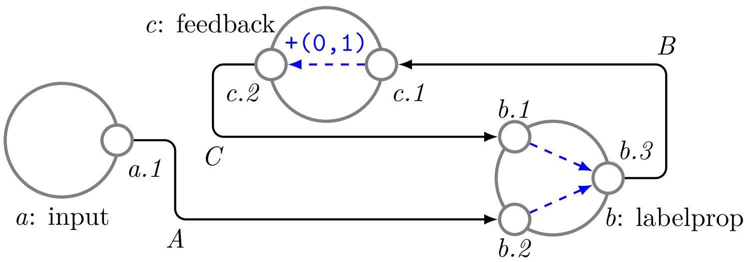

Figure 2 A timely dataflow that computes weakly connected components.

Progress tracking Progress tracking computes frontiers in two steps. A distributed component *exchanges* pointstamp changes (Sections 3 and 4) to construct an approximate, conservative view of all the pointstamps present in the system. Workers use this global view to locally *propagate* changes on the dataflow graph (Section 5) and update the frontiers at the operator input ports. The combined protocol (Section 6) asynchronously executes these two components.

▶ **Running Example** (Weakly Connected Components by Propagating Labels)**.** Figure 2 shows a dataflow that computes weakly connected components (WCC) by assigning integer labels to vertices in a graph, and propagating the lowest label seen so far by each vertex to all its neighbors. The input graph is initially sent by operator *a* as a stream of edges `(s,d)` with timestamp `(0,0)`. Each input port has an associated sharding function to determine which data should be sent to which operator instance: port *b.2* shards the incoming edges `(s,d)` by `s`.

The input operator *a* will continue sending additional edges in the graph as they appear, using increasing timestamps by incrementing one coordinate: `(1,0)`, `(2,0)`, etc. The computation is tasked with reacting to these changes and performing incremental re-computation to produce correct output for each of these input graph versions. The first timestamp coordinate represents logical consistency boundaries for the input and output of the program. We will use the second timestamp coordinate to track the progress of the unbounded iterative algorithm.

The operator *a* starts with a pointstamp (*a.1*, `(0,0)`) on port *a.1*, representing its intent to send data with that timestamp through the connected channel. When it sends messages on channel *A*, these are represented by pointstamps on the port *b.2*; e.g., (*b.2*, `(0,0)`) for the initial timestamp `(0,0)`. When it ceases sending data for a certain timestamp, e.g., `(0,0)`, operator *a* drops the corresponding pointstamp on port *a.1*. The frontier at *b.2* reflects whether pointstamps with a certain timestamp are present at either *a.1* or *b.2*: when they both become absent (when all messages are delivered) each instance of *b* notices that its frontier has advanced and determines it has received its entire share of the input (the graph) for a timestamp.

Each instance of *b* starts with a pointstamp on *b.3* at timestamp `(0,0)`; when it has received its entire share of the input, for each vertex with label `x` and each of its neighbors `n`, it sends `(n,x)` at timestamp `(0,0)`. This stream then traverses operator *c*, that increases the timestamp associated to each message by `(0,1)`, and reaches port *b.1*, which shards the incoming tuples `(n,x)` by `n`. Operator *b* inspects the frontier on *b.1* to determine when it has received all messages with timestamp `(0,1)`. These messages left *b.3* with timestamp `(0,0)`. The progress tracking mechanism will correctly report the frontier at *b.1* by taking into consideration the summary between *c.1* and *c.2*.

Operator *b* collects all label updates from *b.1* and, for those vertices that received a value that is smaller than the current label, it updates internal state and sends a new update via *b.3* with timestamp `(0,1)`. This process then repeats with increasing timestamps, `(0,2)`, `(0,3)`, etc., for each trip around the loop, until ultimately no new update message is generated on port *b.3* by any of the operator instances, for a certain family of timestamps (t_1, t_2) with a fixed t_1 corresponding to the input version being considered. Operator b determines it has correctly labeled all connected components for a given t_1 when the frontier at *b.1* does not contain a (t_1, t_2) such that $t_2 \preceq$ the graph's diameter. In practice, once operator b determines it has computed the output for a given t_1, the operator would also send the output on an additional outgoing channel to deliver it to the user. Later, operator b continues processing for further input versions, indicated by increasing t_1, with timestamps $(t_1,0)$, $(t_1,1)$, etc. ◀

3 The Clocks Protocol

In this section, we present Abadi et al.'s approach to modeling the distributed component of progress tracking [4], termed the *clocks protocol*. Instead of showing their TLA$^+$ Proof System formalization, we present our re-formalization of the protocol in Isabelle. Thereby, this section serves as an introduction to both the protocol and the relevant Isabelle constructs.

The clocks protocol is a distributed algorithm to track existing pointstamps in a dataflow. It models a finite set of workers. Each worker stores a (finite) multiset of pointstamps as seen from its perspective and shares updates to this information with all other workers. The protocol considers workers as black boxes, i.e., it does *not* model their dataflow graph, locations, and timestamps. We extend the protocol to take these components into account in Section 5.

In Isabelle, we use the type variable $'w :: \mathit{finite}$ to represent workers. We assume that $'w$ belongs to the *finite* type class, which assures that $'w$'s universe is finite. Similarly, we model pointstamps abstractly by $'p :: \mathit{order}$. The *order* type class assumes the existence of a partial order $\leqslant\, :: {'p} \Rightarrow {'p} \Rightarrow \mathit{bool}$ (and the corresponding strict order $<$).

We model the protocol as a transition system that acts on configurations given as follows:

record $({'w} :: \mathit{finite}, {'p} :: \mathit{order})\ \mathit{conf} =$
  rec :: $'p\ \mathit{zmset}$
  msg :: $'w \Rightarrow {'w} \Rightarrow {'p}\ \mathit{zmset\ list}$
  temp :: $'w \Rightarrow {'p}\ \mathit{zmset}$
  glob :: $'w \Rightarrow {'p}\ \mathit{zmset}$

Here, rec c denotes the global multiset of pointstamps (or records) that are present in a system's configuration c. We use the type $'p\ \mathit{zmset}$ of *signed multisets* [8]. An element $M :: {'p}\ \mathit{zmset}$ can be thought of as a function of type $'p \Rightarrow \mathit{int}$, which is non-zero only for finitely many values. (In contrast, an unsigned multiset $M :: {'p}\ \mathit{mset}$ corresponds to a function of type $'p \Rightarrow \mathit{nat}$.) Signed multisets enjoy nice algebraic properties; in particular, they form a group. This significantly simplifies the reasoning about subtraction. However, rec c will always store only non-negative pointstamp counts. The other components of a configuration c are

- the progress message queues msg $c\ w\ w'$, which denote the progress update messages sent from worker w to worker w' (not to be confused with data messages, which are accounted for in rec c but do not participate in the protocol otherwise);
- the temporary changes temp $c\ w$ in which worker w stores changes to pointstamps that it might need to communicate to other workers; and
- the local approximation glob $c\ w$ of rec c from the perspective of worker w (we use Abadi et al. [4]'s slightly misleading term glob for the worker's *local* view on the global state).

In contrast to rec, these components may contain a negative count $-i$ for a pointstamp p, which denotes that i occurrences of p have been discarded.

The following predicate characterizes the protocol's initial configurations. We write $\{\#\}_z$ for the empty signed multiset and $M \#_z p$ for the count of pointstamp p in a signed multiset M.

$$\textbf{definition}\ \mathsf{Init} :: ('w, 'p)\ \mathit{conf} \Rightarrow \mathit{bool}\ \textbf{where}$$
$$\mathsf{Init}\ c = (\forall p.\ \mathsf{rec}\ c\ \#_z\ p \geqslant 0) \wedge (\forall w\ w'.\ \mathsf{msg}\ c\ w\ w' = [\,]) \wedge (\forall w.\ \mathsf{temp}\ c\ w = \{\#\}_z) \wedge (\forall w.\ \mathsf{glob}\ c\ w = \mathsf{rec}\ c)$$

In words: all global pointstamp counts in rec must be non-negative and equal to each worker's local view glob; all message queues and temporary changes must be empty.

Referencing our WCC example described in Section 2, the clocks protocol is the component in charge of distributing pointstamp changes to other workers. When one instance of the input operator a ceases sending data for a certain family of timestamps $(t_1, 0)$ it drops the corresponding pointstamp: the clocks protocol is in charge of *exchanging* this information with other workers, so that they can determine when all instances of a have ceased producing messages for a certain timestamp. This happens for all pointstamp changes in the system, including pointstamps that represent messages in-flight on channels.

The configurations evolve via one of three actions:

perf_op: A worker may perform an operation that causes a change in pointstamps. Changes may remove certain pointstamps and add others. They are recorded in rec and temp.

send_upd: A worker may broadcast some of its changes stored in temp to all other workers.

recv_upd: A worker may receive an earlier broadcast and update its local view glob.

Overall, the clocks protocol aims to establish that glob is a safe approximation for rec. Safe means here that no pointstamp in rec is less than any of glob's minimal pointstamps. To achieve this property, the protocol imposes a restriction on which new pointstamps may be introduced in rec and which progress updates may be broadcast. This restriction is the *uprightness* property that ensures that a pointstamp can only be introduced if simultaneously a smaller (supporting) pointstamp is removed. Formally, a signed multiset of pointstamps is upright if every positive entry is accompanied by a smaller negative entry:

$$\textbf{definition}\ \mathsf{supp} :: 'p\ \mathit{zmset} \Rightarrow 'p \Rightarrow \mathit{bool}\ \textbf{where}\ \mathsf{supp}\ M\ p = (\exists p' < p.\ M\ \#_z\ p' < 0)$$
$$\textbf{definition}\ \mathsf{upright} :: 'p\ \mathit{zmset} \Rightarrow \mathit{bool}\ \textbf{where}\ \mathsf{upright}\ M = (\forall p.\ M\ \#_z\ p > 0 \longrightarrow \mathsf{supp}\ M\ p)$$

Abadi et al. [4] additionally require that the pointstamp p' in supp's definition satisfies $\forall p'' \leqslant p'.\ M\ \#_z\ p'' \leqslant 0$. The two variants of upright are equivalent in our formalization because signed multisets are finite and thus minimal elements exist even without $\leqslant$ being well-founded. The extra assumption on p' is occasionally useful in proofs.

In practice, uprightness means that operators are only allowed to transition to pointstamps forward in time, and cannot re-introduce pointstamps that they relinquished. This is necessary to ensure that the frontiers always move to later timestamps and remain a conservative approximation of the pointstamps still present in the system. An advancing frontier triggers computation in some of the dataflow operators, for example to output the result of a time-based aggregation: this should only happen once all the relevant incoming data has been processed. This is the intuition behind the safety property of the protocol, Safe, discussed later in this section.

Figure 3 defines the three protocol actions formally as transition relations between an old configuration c and a new configuration c' along with the definition of the overall transition relation Next, which in addition to performing one of the actions may stutter, i.e., leave $c' = c$ unchanged. The three actions take further parameters as arguments, which we explain next.

The action perf_op is parameterized by a worker w and two (unsigned) multisets Δ_{neg} and Δ_{pos}, corresponding to negative and positive pointstamp changes. The action's overall effect on the pointstamps is thus $\Delta = \Delta_{pos} - \Delta_{neg}$. Here and elsewhere, subtraction expects signed multisets as arguments and we omit the type conversions from unsigned to signed multisets (which are included in our Isabelle formalization). The action is only enabled if its parameters

$$\textbf{definition}\ \mathsf{perf\_op} :: {}'w \Rightarrow {}'p\ mset \Rightarrow {}'p\ mset \Rightarrow ({}'w, {}'p)\ conf \Rightarrow ({}'w, {}'p)\ conf \Rightarrow bool\ \textbf{where}$$
$$\mathsf{perf\_op}\ w\ \Delta_{neg}\ \Delta_{pos}\ c\ c' = \underline{\mathsf{let}}\ \Delta = \Delta_{pos} - \Delta_{neg}\ \underline{\mathsf{in}}\ (\forall p.\ \Delta_{neg}\#p \leqslant \mathsf{rec}\ c\#_z p) \wedge \mathsf{upright}\ \Delta \wedge$$
$$c' = c(\!|\mathsf{rec} = \mathsf{rec}\ c + \Delta,\ \mathsf{temp} = (\mathsf{temp}\ c)(w := \mathsf{temp}\ c\ w + \Delta)|\!)$$

$$\textbf{definition}\ \mathsf{send\_upd} :: {}'w \Rightarrow {}'p\ set \Rightarrow ({}'w, {}'p)\ conf \Rightarrow ({}'w, {}'p)\ conf \Rightarrow bool\ \textbf{where}$$
$$\mathsf{send\_upd}\ w\ P\ c\ c' = \underline{\mathsf{let}}\ \gamma = \{\#p \in\#_z \mathsf{temp}\ c\ w.\ p \in P\#\}\ \underline{\mathsf{in}}$$
$$\gamma \neq \{\#\}_z \wedge \mathsf{upright}\ (\mathsf{temp}\ c\ w - \gamma) \wedge$$
$$c' = c(\!|\mathsf{msg} = (\mathsf{msg}\ c)(w := \lambda w'.\ \mathsf{msg}\ c\ w\ w' \cdot [\gamma]),\ \mathsf{temp} = (\mathsf{temp}\ c)(w := \mathsf{temp}\ c\ w - \gamma)|\!)$$

$$\textbf{definition}\ \mathsf{recv\_upd} :: {}'w \Rightarrow {}'w \Rightarrow ({}'w, {}'p)\ conf \Rightarrow ({}'w, {}'p)\ conf \Rightarrow bool\ \textbf{where}$$
$$\mathsf{recv\_upd}\ w\ w'\ c\ c' = \mathsf{msg}\ c\ w\ w' \neq [] \wedge$$
$$c' = c(\!|\mathsf{msg} = (\mathsf{msg}\ c)(w := (\mathsf{msg}\ c\ w)(w' := \mathsf{tl}\ (\mathsf{msg}\ c\ w\ w'))),$$
$$\mathsf{glob} = (\mathsf{glob}\ c)(w' := \mathsf{glob}\ c\ w' + \mathsf{hd}\ (\mathsf{msg}\ c\ w\ w'))|\!)$$

$$\textbf{definition}\ \mathsf{Next} :: ({}'w, {}'p)\ conf \Rightarrow ({}'w, {}'p)\ conf \Rightarrow bool\ \textbf{where}$$
$$\mathsf{Next}\ c\ c' = (c = c') \vee (\exists w\ \Delta_{neg}\ \Delta_{pos}.\ \mathsf{perf\_op}\ w\ \Delta_{neg}\ \Delta_{pos}\ c\ c') \vee$$
$$(\exists w\ P.\ \mathsf{send\_upd}\ w\ P\ c\ c') \vee (\exists w\ w'.\ \mathsf{recv\_upd}\ w\ w'\ c\ c')$$

Figure 3 Transition relation of Abadi et al.'s clocks protocol

satisfy two requirements. First, only pointstamps present in rec may be dropped, and thus the counts from Δ_{neg} must be bounded by the ones from rec. (Arguably, accessing rec is problematic for distributed workers. We rectify this modeling deficiency in Section 4.) Second, Δ must be upright, which ensures that we will never introduce a pointstamp that is lower than any pointstamp in rec. If these requirements are met, the action can be performed and will update both rec and temp with Δ (expressed using Isabelle's record and function update syntax).

The action send_upd is parameterized by a worker (sender w) and a set of pointstamps P, the outstanding changes to which, called γ, we want to broadcast. The key requirement is that the still unsent changes remain upright. Note that it is always possible to send all changes or all positive changes in temp, because any multiset without a positive change is upright. The operation enqueues γ in all message queues that have w as the sender. We model first-in-first-out queues as lists, where enqueuing means appending at the end ($\_ \cdot [\_]$).

Finally, the action recv_upd is parameterized by two workers (sender w and receiver w'). Given a non-empty queue msg $c\ w\ w'$, the action dequeues the first message (head hd gives the message, tail tl the queue's remainder) and adds it to the receiver's glob.

An execution of the clocks protocol is an infinite sequence of configurations. Infinite sequences of elements of type $'a$ are expressed in Isabelle using the coinductive datatype (short codatatype) of streams defined as **codatatype** $'a\ stream$ = Stream $'a$ ($'a\ stream$). We can inspect a stream's head and tail using the functions shd :: $'a\ stream \Rightarrow {}'a$ and stl :: $'a\ stream \Rightarrow {}'a\ stream$. Valid protocol executions satisfy the predicate Spec, i.e., they start in an initial configuration and all neighboring configurations are related by Next:

$$\textbf{definition}\ \mathsf{Spec} :: ({}'w, {}'p)\ conf\ stream \Rightarrow bool\ \textbf{where}$$
$$\mathsf{Spec}\ s = \mathsf{now}\ \mathsf{Init}\ s \wedge \mathsf{alw}\ (\mathsf{relates}\ \mathsf{Next})\ s$$

The operators now and relates lift unary and binary predicates over configurations to executions by evaluating them on the first one or two configurations respectively: now $P\ s$ = P (shd s) and relates $R\ s = R$ (shd s) (shd (stl s)). The coinductive operator alw resembles a temporal logic operator: alw $P\ s$ holds if P holds for all suffixes of s.

$$\textbf{coinductive}\ \mathsf{alw} :: ({}'a\ stream \Rightarrow bool) \Rightarrow {}'a\ stream \Rightarrow bool\ \textbf{where}$$
$$P\ s \longrightarrow \mathsf{alw}\ P\ (\mathsf{stl}\ s) \longrightarrow \mathsf{alw}\ P\ s$$

We use the operators now, relates, and alw not only to specify valid execution, but also to state the main safety property. Moreover, we use the predicate vacant to express that a pointstamp (and all smaller pointstamps) are not present in a signed multiset:

definition vacant :: $'p\ zmset \Rightarrow\ 'p \Rightarrow bool$ **where** vacant $M\ p = (\forall p' \leqslant p.\ M \#_z\ p' = 0)$

Safety states that if any worker's glob becomes vacant up to some pointstamp, then that pointstamp and any lesser ones do not exist in the system, i.e., are not present in rec (and will remain so). Thus, safety allows workers to learn locally, via glob, something about the system's global state rec, namely that they will never encounter certain pointstamps again. Formally:

definition Safe :: $('w,\ 'p)\ conf\ stream \Rightarrow bool$ **where**
Safe $s = (\forall w\ p.$ now $(\lambda c.$ vacant (glob $c\ w)\ p)\ s \longrightarrow$ alw (now $(\lambda c.$ vacant (rec $c)\ p)\ s))$

lemma *safe:* Spec $s \longrightarrow$ alw Safe s

Proof Sketch. We prove safety following Abadi et al. [4]. First, we establish three invariants by showing that Next preserves them:

1. rec only contains positive entries
2. rec is the sum of any worker w's glob and its *incoming information* info $c\ w = \sum_{w'}($temp $c\ w' + \sum_{M \in \text{set (msg } c\ w'\ w)} M)$, that is the sum of all workers' temp and all msg directed towards w
3. any worker w's incoming information is upright

We then show that whenever rec becomes vacant up to some pointstamp p, then it forever stays vacant up to p. Thus, we can eliminate the "inner" occurrence of alw from the definition of Safe. The remaining property follows by contradiction, i.e., by assuming a non-zero count for some pointstamp p in rec, up to which some worker w's glob is vacant. Invariants 1 and 2 imply that w's incoming information has a positive count for p. Because it is upright by invariant 3, w's incoming information must also contain a smaller pointstamp $q < p$ with a negative count. But w's glob count for q must be zero (recall that w's glob is vacant up to p), which together with invariant 2 implies that rec has a negative count at q. This contradicts invariant 1. ◀

Having established safety, we also prove a second important property of glob formalized by Abadi et al.: monotonicity. This property states that once glob becomes vacant upto some pointstamp p, it will forever stay so:

definition Mono :: $('w,\ 'p)\ conf \Rightarrow ('w,\ 'p)\ conf \Rightarrow bool$ **where**
Mono $c\ c' = (\forall w\ p.$ vacant (glob $c\ w)\ p \longrightarrow$ vacant (glob $c'\ w)\ p)$

lemma *mono:* Spec $s \longrightarrow$ alw (relates Mono) s

Establishing glob's monotonicity is significantly more difficult than proving the same property for rec, which we have used in the proof of safety. New positive entries in rec can only be introduced in the perf_op transition, where they are guarded by smaller negative changes due to the uprightness requirement. In contrast, glob is altered in the recv_upd transition, where it is far less clear a priori why this step cannot introduce pointstamps up to which glob is vacant. The key idea, again due to Abadi et al., to establish glob's monotonicity is to generalize the notion of uprightness and show that all individual messages from msg satisfy the generalized notion. Abadi et al. call the generalized notion *beta uprightness*. It allows positive pointstamp entries from a message $M ::\ 'p\ zmset$ to be supported not only by smaller negative pointstamp entries in M itself, but also by negative entries in another multiset $N ::\ 'p\ zmset$.

definition beta_upright :: $'p\ zmset \Rightarrow\ 'p\ zmset \Rightarrow bool$ **where**
beta_upright $M\ N = (\forall p.\ M \#_z\ p > 0 \longrightarrow (\exists p' < p.\ M \#_z\ p' < 0 \lor N \#_z\ p' < 0))$

We do not describe in detail how beta uprightness helps with monotonicity, but the main step is to establish the invariant that all messages M from msg are beta upright with respect to N being the sum of messages following M in msg and the sender's temp.

Overall, we have replicated the formalization of Abadi et al.'s clocks protocol and proofs of its safety and the monotonicity of glob, each worker's approximated view of the system's pointstamps. Their protocol accurately models the implementation of the progress tracking protocol's distributed component in Timely Dataflow's original implementation Naiad with one subtle exception. The Naiad API (`OnNotify`, `SendBy`) allows an operator to repeatedly send data messages through its output port, which generates pointstamps at the receiver, without requiring that a pointstamp on the output port is decremented. This can result in a perf_op transition that is not upright.[2] Additionally, the modern reference implementation of Timely Dataflow in Rust is more expressive than Naiad, and permits multiple operations that result in non-upright changes. We address and correct this limitation of the clocks protocol in Section 4.

One example of an operator that expresses behavior that results in non-upright changes is the input operator a in the WCC example. This operator may be reading data from an external source, and as soon as it receives new edges, it can forward them with the current pointstamp ($a.1$, `(`t_1`,0)`). This operator may be invoked multiple times, and perform this action repeatedly, until it determines from the external source that it should mark a certain timestamp as complete by dropping the pointstamp. All of these intermediate actions that send data at `(`t_1`,0)` are not upright, as sending messages creates new pointstamps on the message targets, without dropping a smaller pointstamp that can support the postive change.

4 Exchanging Progress

As outlined in the previous section, the clocks protocol is not flexible enough to capture executions with non-upright changes, which are desired and supported by concrete implementations of Timely Dataflow. At the same time, the protocol captures behaviors that are not reasonable in practice. Specifically, the clocks protocol does not separate the worker-local state from the system's global state. The perf_op transition, which is meant to be executed by a single worker, uses the global state to check whether the transition is enabled and simultaneously updates the global state rec as part of the transition. In particular, a single perf_op transition allows a worker to drop a pointstamp that in the real system "belongs" to a different worker w and simultaneously consistently updates w's state. In concrete implementations of Timely Dataflow, workers execute perf_op's asynchronously, and thus can only base the transition on information that is locally available to them.

Our modified model of the protocol, called *exchange*, resolves both issues. As the first step, we split the rec field into worker-local signed multisets caps of pointstamps, which we call *capabilities* as they indicate the possibility for the respective worker to emit these pointstamps. Workers may transfer capabilities to other workers. To do so, they asynchronously send capabilities as data messages to a central multiset data of pairs of workers (receivers) and pointstamps. We arrive at the following updated type of configurations:

[2] We refer here to locations as presented in Section 2. The model in Naiad is slightly different: there is no notion of ports, and pointstamp locations are either operators or edges. A straightforward translation of the Naiad model interprets pointstamps on operators as pointstamps on their source port, and pointstamps on edges become pointstamps on the associated target ports.

$$
\begin{array}{l}
\textbf{definition } \mathsf{recv\_cap} :: {}'w \Rightarrow {}'p \Rightarrow ({}'w, {}'p)\ \mathit{conf} \Rightarrow ({}'w, {}'p)\ \mathit{conf} \Rightarrow \mathit{bool}\ \textbf{where} \\
\quad \mathsf{recv\_cap}\ w\ p\ c\ c' = (w, p) \in_\#\ \mathsf{data}\ c \wedge \\
\qquad c' = c(\!|\mathsf{caps} = (\mathsf{caps}\ c)(w := \mathsf{caps}\ c\ w + \{\#p\#\}_z),\ \mathsf{data} = \mathsf{data}\ c - \{\#(w, p)\#\}|\!)
\end{array}
$$

$$
\begin{array}{l}
\textbf{definition } \mathsf{perf\_op} :: {}'w \Rightarrow {}'p\ \mathit{mset} \Rightarrow ({}'w \times {}'p)\ \mathit{mset} \Rightarrow {}'p\ \mathit{mset} \Rightarrow \\
\qquad ({}'w, {}'p)\ \mathit{conf} \Rightarrow ({}'w, {}'p)\ \mathit{conf} \Rightarrow \mathit{bool}\ \textbf{where} \\
\quad \mathsf{perf\_op}\ w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}\ c\ c' = \\
\qquad (\Delta_{data} \neq \{\#\} \vee \Delta_{self} - \Delta_{neg} \neq \{\#\}_z) \wedge (\forall p.\ \Delta_{neg}\ \#\ p \leqslant \mathsf{caps}\ c\ w\ \#_z\ p) \wedge \\
\qquad (\forall (w', p) \in_\#\ \Delta_{data}.\ \exists p' < p.\ \mathsf{caps}\ c\ w\ \#_z\ p' > 0) \wedge \\
\qquad (\forall p \in_\#\ \Delta_{self}.\ \exists p' \leqslant p.\ \mathsf{caps}\ c\ w\ \#_z\ p' > 0) \wedge \\
\qquad c' = c(\!|\mathsf{caps} = (\mathsf{caps}\ c)(w := \mathsf{caps}\ c\ w + \Delta_{self} - \Delta_{neg}),\ \mathsf{data} = \mathsf{data}\ c + \Delta_{data}, \\
\qquad\qquad \mathsf{temp} = (\mathsf{temp}\ c)(w := \mathsf{temp}\ c\ w + (\mathsf{snd}\ `\#\ \Delta_{data} + \Delta_{self} - \Delta_{neg}))|\!)
\end{array}
$$

$$
\begin{array}{l}
\textbf{definition } \mathsf{send\_upd} :: {}'w \Rightarrow {}'p\ \mathit{set} \Rightarrow ({}'w, {}'p)\ \mathit{conf} \Rightarrow ({}'w, {}'p)\ \mathit{conf} \Rightarrow \mathit{bool}\ \textbf{where} \\
\quad \mathsf{send\_upd}\ w\ P\ c\ c' = \underline{\mathsf{let}}\ \gamma = \{\#p \in_{\#_z} \mathsf{temp}\ c\ w.\ p \in P\#\}\ \underline{\mathsf{in}} \\
\qquad \gamma \neq \{\#\}_z \wedge \mathsf{justified}\ (\mathsf{caps}\ c\ w)\ (\mathsf{temp}\ c\ w - \gamma) \wedge \\
\qquad c' = c(\!|\mathsf{msg} = (\mathsf{msg}\ c)(w := \lambda w'.\ \mathsf{msg}\ c\ w\ w' \cdot [\gamma]),\ \mathsf{temp} = (\mathsf{temp}\ c)(w := \mathsf{temp}\ c\ w - \gamma)|\!)
\end{array}
$$

$$
\begin{array}{l}
\textbf{definition } \mathsf{recv\_upd} :: {}'w \Rightarrow {}'w \Rightarrow ({}'w, {}'p)\ \mathit{conf} \Rightarrow ({}'w, {}'p)\ \mathit{conf} \Rightarrow \mathit{bool}\ \textbf{where} \\
\quad \mathsf{recv\_upd}\ w\ w'\ c\ c' = \mathsf{msg}\ c\ w\ w' \neq [] \wedge \\
\qquad c' = c(\!|\mathsf{msg} = (\mathsf{msg}\ c)(w := (\mathsf{msg}\ c\ w)(w' := \mathsf{tl}\ (\mathsf{msg}\ c\ w\ w'))), \\
\qquad\qquad \mathsf{glob} = (\mathsf{glob}\ c)(w' := \mathsf{glob}\ c\ w' + \mathsf{hd}\ (\mathsf{msg}\ c\ w\ w'))|\!)
\end{array}
$$

$$
\begin{array}{l}
\textbf{definition } \mathsf{Next} :: ({}'w, {}'p)\ \mathit{conf} \Rightarrow ({}'w, {}'p)\ \mathit{conf} \Rightarrow \mathit{bool}\ \textbf{where} \\
\quad \mathsf{Next}\ c\ c' = (c = c') \vee (\exists w\ p.\ \mathsf{recv\_cap}\ w\ p\ c\ c') \vee \\
\qquad (\exists w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}.\ \mathsf{perf\_op}\ w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}\ c\ c') \vee \\
\qquad (\exists w\ P.\ \mathsf{send\_upd}\ w\ P\ c\ c') \vee (\exists w\ w'.\ \mathsf{recv\_upd}\ w\ w'\ c\ c')
\end{array}
$$

Figure 4 Transition relation of the exchange protocol

$$
\begin{array}{l}
\textbf{record}\ ({}'w :: \mathit{finite}, {}'p :: \mathit{order})\ \mathit{conf} = \\
\quad \mathsf{caps} :: {}'w \Rightarrow {}'p\ \mathit{zmset} \\
\quad \mathsf{data} :: ({}'w \times {}'p)\ \mathit{mset} \\
\quad \mathsf{msg} :: {}'w \Rightarrow {}'w \Rightarrow {}'p\ \mathit{zmset}\ \mathit{list} \\
\quad \mathsf{temp} :: {}'w \Rightarrow {}'p\ \mathit{zmset} \\
\quad \mathsf{glob} :: {}'w \Rightarrow {}'p\ \mathit{zmset}
\end{array}
$$

Including this fine-grained view on pointstamps will allow workers to make transitions based on worker-local information. The entirety of the system's pointstamps, rec, which was previously part of the configuration and which the protocol aims to track, can be computed as the sum of all the workers' capabilities and data's in-flight pointstamps.

$$\textbf{definition } \mathsf{rec} :: ({}'w, {}'p)\ \mathit{conf} \Rightarrow {}'p\ \mathit{zmset}\ \textbf{where}\ \mathsf{rec}\ c = (\textstyle\sum_w \mathsf{caps}\ c\ w) + \mathsf{snd}\ `\#\ \mathsf{data}\ c$$

Here, the infix operator $`\#$ denotes the image of a function over a multiset with resulting counts given by $(f\ `\#\ M)\ \#\ x = \sum_{y \in \{y \in \# M \mid f\ y = x\}} M\ \#\ y$.

The exchange protocol's initial state allows workers to start with some positive capabilities. Each worker's glob must correctly reflect all initially present capabilities.

$$
\begin{array}{l}
\textbf{definition } \mathsf{Init} :: ({}'w, {}'p)\ \mathit{conf} \Rightarrow \mathit{bool}\ \textbf{where} \\
\quad \mathsf{Init}\ c = (\forall w\ p.\ \mathsf{caps}\ c\ w\ \#_z\ p \geqslant 0) \wedge \mathsf{data}\ c = \{\#\} \wedge \\
\qquad (\forall w\ w'.\ \mathsf{msg}\ c\ w\ w' = [\,]) \wedge (\forall w.\ \mathsf{temp}\ c\ w = \{\#\}_z) \wedge (\forall w.\ \mathsf{glob}\ c\ w = \mathsf{rec}\ c)
\end{array}
$$

The transition relation of the exchange protocol, shown in Figure 4, is similar to that of the clocks protocol. We focus on the differences between the two protocols. First, the exchange protocol has an additional transition recv_cap to receive a previously sent capability. The transition removes a pointstamp from data and adds it to the receiving worker's capabilities.

The perf_op transition resembles its homonymous counterpart from the clocks protocol. Yet, the information flow is more fine grained. In particular, the transition is parameterized by a worker w and three multisets of pointstamps. As in the clocks protocol, the multiset Δ_{neg} represents negative changes to pointstamps. Only pointstamps for which w owns a capability in caps may be dropped in this way. The other two multisets Δ_{data} and Δ_{self} represent positive changes. The multiset Δ_{data} represents positive changes to other workers' capabilities—the receiving worker is stored in Δ_{data}. These changes are not immediately applied to the other worker's caps, but are sent via the data field. The multiset Δ_{self} represents positive changes to w's capabilities, which are applied immediately applied to w's caps. The separation between Δ_{data} and Δ_{self} is motivated by different requirements on these positive changes to pointstamps that we prove to be sufficient for safety. To send a positive capability to another worker, w is required to hold a positive capability for a strictly smaller pointstamp. In contrast, w can create a new capability for itself, if it is already holding a capability for the very same (or a smaller) pointstamp. In other words, w can arbitrarily increase the multiset counts of its own capabilities. Note that, unlike in the clocks protocol, there is no requirement of uprightness and, in fact, workers are not required to perform negative changes at all. Of course, it is useful for workers to perform negative changes every now and then so that the overall system can make progress.

The first condition in perf_op, namely $\Delta_{data} \neq \{\#\} \vee \Delta_{self} - \Delta_{neg} \neq \{\#\}_z$, ensures that the transition changes the configuration. In the exchange protocol, we also include explicit stutter steps in the Next relation ($c = c'$) but avoid them in the individual transitions.

Sending (send_upd) and receiving (recv_upd) progress updates works precisely as in the clocks protocol except for the condition on what remains in the sender's temp highlighted in gray in Figure 4. Because we allowed perf_op to perform non-upright changes, we can no longer expect the contents of temp to be upright. Instead, we use the predicate justified, which offers three possible justifications for positive entries in the signed multiset M (in contrast to upright's sole justification of being supported in M):

definition justified :: $'p\ zmset \Rightarrow\ 'p\ zmset \Rightarrow bool$ **where**

$$\text{justified}\ C\ M = (\forall p.\ M \#_z p > 0 \longrightarrow \text{supp}\ M\ p \vee (\exists p' < p.\ C \#_z p' > 0) \vee M \#_z p < C \#_z p)$$

Thus, a positive count for pointstamp p in M may be either

- supported in M, i.e., in particular every upright change is justified, or
- justified by a smaller pointstamp in C, which we think of as the sender's capabilities, or
- justified by p in C, with the requirement that p's count in M is smaller than p's count in C.

The definitions of valid executions Spec and the safety predicate Safe are unchanged compared to the clocks protocol. Also, we prove precisely the same safety property *safe* following a similar proof structure. The main difference is that uprightness invariant 3 is replaced by the statement that every worker's incoming information is justified with respect to pointstamps present in rec, i.e. $\forall w.$ justified (rec c) (info c w). It is more tedious to reason about pointstamps being justified compared with being upright due to the three-way case distinction that is usually necessary. These case distinctions occur when establishing the above invariant, but also in the contradiction proof establishing safety. The contradiction proof proceeds as before by assuming a non-zero count for some pointstamp p in rec, up to which some worker w's glob is vacant. Crucially, p is now additionally and without loss of generality assumed to be a minimal pointstamp with this property. By invariants 1 and 2, we deduce that w's incoming information has a positive count for p. Because it is justified by the new invariant 3, we perform the

$$
\begin{array}{l}
\textbf{locale}\ \mathsf{graph} = \\
\quad \textbf{fixes}\ \mathsf{weights} :: ('vtx :: \mathit{finite}) \Rightarrow 'vtx \Rightarrow ('lbl :: \{order,\ monoid\_add\})\ \mathit{antichain} \\
\quad \textbf{assumes}\ (l :: 'lbl) \geqslant 0 \quad \textbf{and}\ (l_1 :: 'lbl) \leqslant l_3 \longrightarrow l_2 \leqslant l_4 \longrightarrow l_1 + l_2 \leqslant l_3 + l_4 \\
\quad \textbf{and}\ \mathsf{weights}\ l\ l = \{\} \\
\textbf{locale}\ \mathsf{dataflow} = \mathsf{graph}\ \mathsf{summary} \\
\quad \textbf{for}\ \mathsf{summary} :: ('l :: \mathit{finite}) \Rightarrow 'l \Rightarrow ('sum :: \{order,\ monoid\_add\})\ \mathit{antichain} + \\
\quad \textbf{fixes}\ \oplus :: ('t :: order) \Rightarrow 'sum \Rightarrow 't \\
\quad \textbf{assumes}\ t \oplus 0 = t \quad \textbf{and}\ (t \oplus s) \oplus s' = t \oplus (s + s') \quad \textbf{and}\ t \leqslant t' \longrightarrow s \leqslant s' \longrightarrow t \oplus s \leqslant t' \oplus s' \\
\quad \textbf{and}\ \mathsf{path}\ l\ l\ xs \longrightarrow xs \neq [\,] \longrightarrow t < t \oplus (\sum xs)
\end{array}
$$

Figure 5 Locales for graphs and dataflows

case distinction on the justification. If p is supported in w's incoming information, we proceed as in the clocks protocol. If p is justified by a positive count for a strictly smaller pointstamp in rec, we obtain a contradiction to p's minimality. Finally, if p's multiplicity in w's incoming information is strictly smaller than p's multiplicity in rec, invariant 2 tells us that p must have a positive count in glob, which contradicts the assumption of glob being vacant up to p.

We prove glob's monotonicity for the exchange protocol, too. The proof resembles the one for the clocks protocol; it requires a generalization of justified, called justified_with, to account for positive entries in every in-flight progress message M. The generalization has the same three disjuncts as justified, but relaxes the first and third disjunct to take into account an additional multiset N of justifying pointstamps. Usages of justified_with instantiate N with the sum of messages following M in msg and the sender's temp.

$$
\begin{array}{l}
\textbf{definition}\ \mathsf{justified\_with} :: 'p\ zmset \Rightarrow 'p\ zmset \Rightarrow 'p\ zmset \Rightarrow bool\ \textbf{where} \\
\quad \mathsf{justified\_with}\ C\ M\ N = (\forall p.\ M \#_z\, p > 0 \longrightarrow \\
\qquad (\exists p' < p.\ M \#_z\, p' < 0 \lor N \#_z\, p' < 0) \lor (\exists p' < p.\ C \#_z\, p' > 0) \lor (M + N) \#_z\, p < C \#_z\, p)
\end{array}
$$

We also derive the following additional property of glob, which shows that any in-flight progress updates to a pointstamp p, positive or negative, have a corresponding positive count for some pointstamp less or equal than p in the receiver's glob. We will use this property when combining in Section 6 the exchange protocol with the worker-local progress propagation, which we cover next in Section 5.

$$
\begin{array}{l}
\textbf{lemma}\ \mathit{glob}\text{:}\ \ \mathsf{Spec}\ s \longrightarrow \mathsf{alw}\ (\mathsf{now}\ (\lambda c.\ \forall w\ w'\ p. \\
\quad (\exists M \in \mathsf{set}\ (\mathsf{msg}\ c\ w\ w').\ p \in \#_z\, M) \longrightarrow (\exists p' \leqslant p.\ \mathsf{glob}\ c\ w'\ \#_z\, p' > 0)))\ s
\end{array}
$$

5 Locally Propagating Progress

The previous sections focused on the progress-relevant communication between workers and abstracted over the actual dataflow that is evaluated by each worker. Next, we refine this abstraction: we model the actual dataflow graph as a weighted directed graph with vertices representing operator input and output ports, termed *locations*. We do not distinguish between source and target locations and thus also not between internal and dataflow edges. Each weight denotes a minimum increment that is performed to a timestamp when it conceptually travels along the corresponding edge from one location to another. On a single worker, progress updates can be communicated locally, so that every operator learns which timestamps it may still receive in the future. We formalize Timely Dataflow's approach for this local communication: the algorithm gradually propagates learned pointstamp changes along dataflow edges to update downstream frontiers.

Figure 5 details our modeling of graphs and dataflows, which uses locales [5] to capture our abstract assumptions on dataflows and timestamps. A locale lets us fix parameters (types and constants) and assume properties about them. In our setting, a weighted directed graph is given by a finite (class *finite*) type $'vtx$ of vertices and a weights function that assigns each pair of vertices a weight. To express weights, we fix a type of labels $'lbl$, which we assume to be partially ordered (class *order*) and to form a monoid (class *monoid_add*) with the monoid operation $+$ and the neutral element 0. We assume that labels are non-negative and that $+$ on labels is monotone with respect to the partial order $\leqslant$. A weight is then an antichain of labels, that is a set of incomparable (with respect to $\leqslant$) labels, which we model as follows:

$$\textbf{typedef } ('t :: order)\ antichain = \{A :: 't\ set.\ \mathsf{finite}\ A \wedge (\forall a \in A.\ \forall b \in A.\ a \nless b \wedge b \nless a)\}$$

We use standard set notation for antichains and omit type conversions from antichains to (signed) multisets. The empty antichain $\{\}$ is a valid weight, too, in which case we think of the involved vertices as not being connected to each other. Thus, the graph locale's final assumption expresses the non-existence of self-edges in a graph.

Within the graph locale, we can define the predicate path :: $'vtx \Rightarrow 'vtx \Rightarrow 'lbl\ list \Rightarrow bool$. Intuitively, path v w xs expresses that the list of labels xs is a valid path from v to w (the empty list being a valid path only if $v = w$ and any weight $l \in$ weights u v can extend a valid path from v to w to a path from u to w). We omit path's formal straightforward inductive definition. Note that even though self-edges are disallowed, cycles in graphs are possible (and desired). In other words, path v v xs can be true for a non-empty list xs.

The second locale, dataflow, has two purposes. First, it refines the generic graph terminology from vertices and labels to locations ($'l$) and summaries ($'sum$), which is the corresponding terminology used in Timely Dataflow. Second, it introduces the type for timestamps $'t$, which is partially ordered (class *order*) and an operation $\oplus$ (read as "results in") that applies a summary to a timestamp to obtain a new timestamp. We chose the asymmetric symbol for the operation to remind the reader that its two arguments have different types, timestamps and summaries. The locale requires the operation $\oplus$ to be well-behaved with respect to the available vocabulary on summaries (0, $+$, and $\leqslant$). Moreover, it requires that proper cycles xs have a path summary $\sum xs$ (defined by iterating $+$) that strictly increments any timestamp t.

Now, consider a function $P :: 'l \Rightarrow 't\ zmset$ that assigns each location a set of timestamps that it currently holds. We are interested in computing a lower bound of timestamps (with respect to the order $\leqslant$) that may arrive at any location for a given P. Timely Dataflow calls antichains that constitute such a lower bound frontiers. Formally, a frontier is the set of minimal incomparable elements that have a positive count in a signed multiset of timestamps.

$$\textbf{definition } \mathsf{antichain\_of} :: 't\ set \Rightarrow 't\ set\ \textbf{where } \mathsf{antichain\_of}\ A = \{x \in A.\ \neg\exists y \in A.\ y < x\}$$
$$\textbf{lift\_definition } \mathsf{frontier} :: 't\ zmset \Rightarrow 't\ antichain\ \textbf{is } \lambda M.\ \mathsf{antichain\_of}\ \{t.\ M \#_z t > 0\}$$

Our frontier of interest, called the implied frontier, at location l can be computed directly for a given function P by adding, for every location l', every (minimal) possible path summary between l' and l, denoted by the antichain path_summary l' l, to every timestamp present at l' and computing the frontier of the result. Formally, we first lift $\oplus$ to signed multisets and antichains. Then, we use the lifted operator $\bigoplus$ to define the implied frontier.

$$\textbf{definition } \bigoplus :: 't\ zmset \Rightarrow 'sum\ antichain \Rightarrow 't\ zmset\ \textbf{where}$$
$$M \bigoplus A = \textstyle\sum_{s \in A} (\lambda t.\ t \oplus s)\ `\#_z\ M$$
$$\textbf{definition } \mathsf{implied\_frontier} :: ('l \Rightarrow 't\ zmset) \Rightarrow 'l \Rightarrow 't\ antichain\ \textbf{where}$$
$$\mathsf{implied\_frontier}\ P\ l = \mathsf{frontier}\ (\textstyle\sum_{l'} (\mathsf{pos}_z\ (P\ l') \bigoplus \mathsf{path\_summary}\ l'\ l))$$

definition change_multiplicity :: $'l \Rightarrow 't \Rightarrow int \Rightarrow ('l, 't)\ conf \Rightarrow ('l, 't)\ conf \Rightarrow bool$ **where**
change_multiplicity $l\ t\ n\ c\ c' = n \neq 0 \wedge (\exists t' \in$ frontier (implications $c\ l$). $t' \leqslant t) \wedge$
$c' = c$⦇pts = (pts c)($l :=$ pts $c\ l$ + replicate $n\ t$),
work = (work c)($l :=$ work $c\ l$ + frontier (pts $c'\ l$) − frontier (pts $c\ l$))⦈

definition propagate :: $'l \Rightarrow 't \Rightarrow ('l, 't)\ conf \Rightarrow ('l, 't)\ conf \Rightarrow bool$ **where**
propagate $l\ t\ c\ c' = t \in\#_z$ work $c\ l \wedge (\forall l'. \forall t' \in\#_z$ work $c\ l'. \neg t' < t) \wedge$
$c' = c$⦇imp = (imp c)($l :=$ imp $c\ l$ + replicate (work $c\ l\ \#_z\ t$) t,
work = $\lambda l'$. <u>if</u> $l = l'$ <u>then</u> $\{\#t' \in\#_z$ work $c\ l.\ t' \neq t\#\}$
<u>else</u> work $c\ l'$ + ((frontier (imp $c'\ l$) − frontier (imp $c\ l$)) $\bigoplus$ summary $l\ l'$)⦈

definition Next :: $('l, 't)\ conf \Rightarrow ('l, 't)\ conf \Rightarrow bool$ **where**
Next $c\ c' = (c = c') \vee (\exists l\ t\ n.$ change_multiplicity $l\ t\ n\ c\ c') \vee (\exists l\ t.$ propagate $l\ t\ c\ c')$

Figure 6 Transition relation of the local progress propagation

Above and elsewhere, given a signed multiset M, we write f ‘$\#_z$ M for the image (as a signed multiset) of f over M and $\mathsf{pos}_z\ M$ for the signed multiset of M's positive entries.

Computing the implied frontier for each location in this way (quadratic in the number of locations) would be too inefficient, especially because we want to frequently supply operators with up-to-date progress information. Instead, we follow the optimized approach implemented in Timely Dataflow: after performing some work and making some progress, operators start pushing relevant updates only to their immediate successors in the dataflow graph. The information gradually propagates and eventually converges to the implied frontier. Despite this local propagation not being a distributed protocol as such, we formalize it for a fixed dataflow in a similar state-machine style as the earlier exchange protocol.

Local propagation uses a configuration consisting of three signed multiset components.

record $('l :: \mathit{finite}, 't :: \{\mathit{monoid\_add}, \mathit{order}\})\ conf =$
pts :: $'l \Rightarrow 't\ zmset$
imp :: $'l \Rightarrow 't\ zmset$
work :: $'l \Rightarrow 't\ zmset$

Following Timely Dataflow terminology, pointstamps pts are the present timestamps grouped by location (the P function from above). The implications imp are the output of the local propagation and contain an over-approximation of the implied frontier (as we will show). Finally, the worklist work is an auxiliary data structure to store not-yet propagated timestamps.

Initially, all implications are empty and worklists consist of the frontiers of the pointstamps.

definition Init :: $('l, 't)\ conf \Rightarrow bool$ **where**
Init $c = (\forall l.$ imp $c\ l = \{\#\}_z \wedge$ work $c\ l$ = frontier (pts $c\ l$))

The propagation proceeds by executing one of two actions shown in Figure 6. The action change_multiplicity constitutes the algorithm's information input: The system may have changed the multiplicity of some timestamp t at location l and can use this action to notify the propagation algorithm of the change. The change value n is required to be non-zero and the affected timestamp t must be witnessed by some timestamp in the implications. Note that the latter requirement prohibits executing this action in the initial state. The action updates the pointstamps according to the declared change. It also updates the worklist, but only if the update of the pointstamps affects the frontier of the pointstamps at l and moreover the worklists are updated merely by the change to the frontier.

The second action, propagate, applies the information for the timestamp t stored in the worklist at a given location l, to the location's implications (thus potentially enabling the first action). It also updates the worklists at the location's immediate successors in the dataflow graph. Again the worklist updates are filtered by whether they affect the frontier (of the implications) and are adjusted by the summary between l and each successor. Importantly, only minimal timestamps (with respect to timestamps in worklists at all locations) may be propagated, which ensures that any timestamp will eventually disappear from all worklists.

The overall transition relation Next allows us to choose between these two actions and a stutter step. Together with Init, it gives rise to the predicate describing valid executions in the standard way: Spec s = now Init s ∧ alw (relates Next) s.

We show that valid executions satisfy a safety invariant. Ideally, we would like to show that for any t with a positive count in pts at location l and for any path summary s between l and some location l', there is a timestamp in the (frontier of the) implications at l' that is less than or equal to $t \oplus s$. In other words, the location l' is aware that it may still encounter timestamp $t \oplus s$. Stated as above, the invariant does not hold, due to the not-yet-propagated progress information stored in the worklists. If some timestamp, however, does not occur in any worklist (formalized by the below work_vacant predicate), we obtain our desired invariant Safe.

definition work_vacant :: ('l, 't) *conf* ⇒ 't ⇒ *bool* **where**
work_vacant $c\ t = (\forall l\ l'\ s\ t'.\ t' \in\#_z$ work $c\ l \longrightarrow s \in$ path_summary $l\ l' \longrightarrow t' \oplus s \nleq t)$
definition Safe :: ('l, 't) *conf stream* ⇒ *bool* **where**
Safe $c = (\forall l\ l'\ t\ s.$ pts $c\ l\ \#_z\ t > 0 \wedge s \in$ path_summary $l\ l' \wedge$ work_vacant $c\ (t \oplus s) \longrightarrow$
$(\exists t' \in$ frontier (imp $c\ l'$). $t' \leqslant t \oplus s))$
lemma *safe:* Spec $s \longrightarrow$ alw (now Safe) s

In our running WCC example, Safe is for example necessary to determine once operator b has received all incoming updates for a certain round of label propagation, which is encoded as a timestamp (t_1, t_2). If a pointstamp at port *b.3* was not correctly reflected in the frontier at *b.1* the operator may incorrectly determine that it has seen all incoming labels for a certain graph node and proceed to the next round of propagation. Safe states, that this cannot happen and all pointstamps are correctly reflected in relevant downstream frontiers.

The safety proof relies on two auxiliary invariants. First, implications have only positive entries. Second, the sum of the implication and the worklist at a given location l is equal to the sum of the frontier of the pointstamps at l and the sum of all frontiers of the implications of all immediate predecessor locations l' (adjusted by the corresponding summary summary $l'\ l$).

While the above safety property is sufficient to prove safety of the combination of the local propagation and the exchange protocol in the next section, we also establish that the computed frontier of the implications converges to the implied frontier. Specifically, the two frontiers coincide for timestamps which are not contained in any of the worklists.

lemma *implied_frontier:* Spec $s \longrightarrow$ alw (now ($\lambda c.$ work_vacant $c\ t \longrightarrow$
$(\forall l.\ t \in$ frontier (imp $c\ l) \longleftrightarrow t \in$ implied_frontier (pts c) l))) s

6 Progress Tracking

We are now ready to combine the two parts presented so far: the between-worker exchange of progress updates (Section 4) and the worker-local progress propagation (Section 5). The combined protocol takes pointstamp changes and determines per-location frontiers at each operator on each worker. It operates on configurations consisting of a single exchange protocol configuration (referred to with the prefix E) and for each worker a local propagation configuration (prefix P) and a Boolean flag indicating whether the worker has been properly initialized.

$$
\begin{aligned}
&\textbf{definition}\ \mathsf{recv\_cap} :: {}'w \Rightarrow {}'l \times {}'t \Rightarrow ({}'w, {}'l, {}'t)\ \mathit{conf} \Rightarrow ({}'w, {}'l, {}'t)\ \mathit{conf} \Rightarrow \mathit{bool}\ \textbf{where}\\
&\quad \mathsf{recv\_cap}\ w\ p\ c\ c' = \mathsf{E.recv\_cap}\ w\ p\ (\mathsf{exch}\ c)\ (\mathsf{exch}\ c') \wedge \mathsf{prop}\ c' = \mathsf{prop}\ c \wedge \mathsf{init}\ c' = \mathsf{init}\ c\\
&\textbf{definition}\ \mathsf{perf\_op} :: {}'w \Rightarrow ({}'l \times {}'t)\ \mathit{mset} \Rightarrow ({}'w \times ({}'l \times {}'t))\ \mathit{mset} \Rightarrow ({}'l \times {}'t)\ \mathit{mset} \Rightarrow\\
&\qquad ({}'w, {}'l, {}'t)\ \mathit{conf} \Rightarrow ({}'w, {}'l, {}'t)\ \mathit{conf} \Rightarrow \mathit{bool}\ \textbf{where}\\
&\quad \mathsf{perf\_op}\ w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}\ c\ c' = \mathsf{E.perf\_op}\ w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}\ (\mathsf{exch}\ c)\ (\mathsf{exch}\ c') \wedge\\
&\qquad \mathsf{prop}\ c' = \mathsf{prop}\ c \wedge \mathsf{init}\ c' = \mathsf{init}\ c\\
&\textbf{definition}\ \mathsf{send\_upd} :: {}'w \Rightarrow ({}'l \times {}'t)\ \mathit{set} \Rightarrow ({}'w, {}'l, {}'t)\ \mathit{conf} \Rightarrow ({}'w, {}'l, {}'t)\ \mathit{conf} \Rightarrow \mathit{bool}\ \textbf{where}\\
&\quad \mathsf{send\_upd}\ w\ P\ c\ c' = \mathsf{E.send\_upd}\ (\mathsf{exch}\ c)\ (\mathsf{exch}\ c')\ w\ P \wedge \mathsf{prop}\ c' = \mathsf{prop}\ c \wedge \mathsf{init}\ c' = \mathsf{init}\ c\\
&\textbf{definition}\ \mathsf{cm\_all} :: ({}'l, {}'t)\ \mathit{P.conf} \Rightarrow ({}'l \times {}'t)\ \mathit{zmset} \Rightarrow ({}'l, {}'t)\ \mathit{P.conf}\ \textbf{where}\\
&\quad \mathsf{cm\_all}\ c\ \Delta = \mathsf{Set.fold}\ (\lambda(l,t)\ c.\ \mathsf{SOME}\ c'.\ \mathsf{P.change\_multiplicity}\ c\ c'\ l\ t\ (\Delta\ \#_z\ (l,t)))\ c\\
&\qquad \{(l,t).\ (l,t) \in\#_z \Delta\}\\
&\textbf{definition}\ \mathsf{recv\_upd} :: {}'w \Rightarrow {}'w \Rightarrow ({}'w, {}'l, {}'t)\ \mathit{conf} \Rightarrow ({}'w, {}'l, {}'t)\ \mathit{conf} \Rightarrow \mathit{bool}\ \textbf{where}\\
&\quad \mathsf{recv\_upd}\ w\ w'\ c\ c' = \mathsf{init}\ c\ w' \wedge \mathsf{E.recv\_upd}\ w\ t\ (\mathsf{exch}\ c)\ (\mathsf{exch}\ c') \wedge\\
&\qquad \mathsf{prop}\ c' = (\mathsf{prop}\ c)(w' := \mathsf{cm\_all}\ (\mathsf{prop}\ c\ w')\ (\mathsf{hd}\ (\mathsf{E.msg}\ (\mathsf{exch}\ c)))) \wedge \mathsf{init}\ c' = \mathsf{init}\ c\\
&\textbf{definition}\ \mathsf{propagate} :: {}'w \Rightarrow ({}'w, {}'l, {}'t)\ \mathit{conf} \Rightarrow ({}'w, {}'l, {}'t)\ \mathit{conf} \Rightarrow \mathit{bool}\ \textbf{where}\\
&\quad \mathsf{propagate}\ w\ c\ c' = \mathsf{exch}\ c' = \mathsf{exch}\ c \wedge \mathsf{init}\ c' = (\mathsf{init}\ c)(w := \mathsf{True}) \wedge\\
&\qquad (\mathsf{Some} \circ \mathsf{prop}\ c') = (\mathsf{Some} \circ \mathsf{prop}\ c)(w := \mathsf{while\_option}\\
&\qquad\quad (\lambda c.\ \exists l.\ \mathsf{P.work}\ c\ l \neq \{\#\}_z)\ (\lambda c.\ \mathsf{SOME}\ c'.\ \exists l\ t.\ \mathsf{P.propagate}\ l\ t\ c\ c')\ (\mathsf{prop}\ c\ w))\\
&\textbf{definition}\ \mathsf{Next} :: ({}'w, {}'l, {}'t)\ \mathit{conf} \Rightarrow ({}'w, {}'l, {}'t)\ \mathit{conf} \Rightarrow \mathit{bool}\ \textbf{where}\\
&\quad \mathsf{Next}\ c\ c' = (c = c') \vee (\exists w\ p.\ \mathsf{recv\_cap}\ w\ p\ c\ c') \vee\\
&\qquad (\exists w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}.\ \mathsf{perf\_op}\ w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}\ c\ c') \vee\\
&\qquad (\exists w\ P.\ \mathsf{send\_upd}\ w\ P\ c\ c') \vee (\exists w\ w'.\ \mathsf{recv\_upd}\ w\ w'\ c\ c') \vee (\exists w.\ \mathsf{propagate}\ w\ c\ c')
\end{aligned}
$$

Figure 7 Transition relation of the combined progress tracker

$$
\begin{aligned}
&\textbf{record}\ ({}'w :: \mathit{finite}, {}'l :: \mathit{finite}, {}'t :: \{\mathit{monoid\_add}, \mathit{order}\})\ \mathit{conf} =\\
&\quad \mathsf{exch} :: ({}'w,\ {}'l \times {}'t)\ \mathit{E.conf}\\
&\quad \mathsf{prop} :: {}'w \Rightarrow ({}'l,\ {}'t)\ \mathit{P.conf}\\
&\quad \mathsf{init} :: {}'w \Rightarrow \mathit{bool}
\end{aligned}
$$

As pointstamps in the exchange protocol, we use pairs of locations and timestamps. To order pointstamps, we use the following *could-result-in* relation, inspired by Abadi and Isard [3].

$$\textbf{definition}\ \ \leqslant_{cri}\ \ \textbf{where}\ (l,t) \leqslant_{cri} (l',t') = (\exists s \in \mathsf{path\_summary}\ l\ l'.\ t \oplus s \leqslant t')$$

As required by the exchange protocol, this definition yields a partial order. In particular, antisymmetry follows from the assumption that proper cycles have a non-zero summary and transitivity relies on the operation $\oplus$ being monotone. Intuitively, $\leqslant_{cri}$ captures a notion of reachability in the dataflow graph: as timestamp t traverses the graph starting at location l, it could arrive at location l', being incremented to timestamp t'. (In Timely Dataflow, an edge's summary represents the minimal increment to a timestamp when it traverses that edge.)

In an initial combined configuration, all workers are not initialized and all involved configurations are initial. Moreover, the local propagation's pointstamps coincide with exchange protocol's glob, which is kept invariant in the combined protocol.

$$
\begin{aligned}
&\textbf{definition}\ \mathsf{Init} :: ({}'w, {}'l, {}'t)\ \mathit{conf} \Rightarrow \mathit{bool}\ \textbf{where}\\
&\quad \mathsf{Init}\ c = (\forall w.\ \mathsf{init}\ c\ w = \mathsf{False}) \wedge \mathsf{E.Init}\ (\mathsf{exch}\ c) \wedge (\forall w.\ \mathsf{P.Init}\ (\mathsf{prop}\ c\ w)) \wedge\\
&\qquad\quad (\forall w\ l\ t.\ \mathsf{P.pts}\ (\mathsf{prop}\ c\ w)\ l\ \#_z\ t = \mathsf{E.glob}\ (\mathsf{exch}\ c)\ w\ \#_z\ (l,t))
\end{aligned}
$$

Figure 7 shows the combined protocol's transition relation Next. Most actions have identical names as the exchange protocol's actions and they mostly perform the corresponding actions on the exchange part of the configuration. In addition, the recv_upd action also performs several change_multiplicity local propagation actions: the receiver updates the state of its local propagation configuration for all received timestamp updates. The action propagate does not have a counterpart in the exchange protocol. It iterates, using the while_option combinator from Isabelle's library, propagation on a single worker until all worklists are empty. The term while_option $b\ c\ s$ repeatedly applies c starting from the initial state s, until the predicate b is satisfied. Overall, it evaluates to Some s' satisfying $\neg b\ s'$ and $s' = c\ (\cdots (c\ s))$ with the least possible number of repetitions of c and to None if no such state exists. Thus, it is only possible to take the propagate action, if the repeated propagation terminates for the considered configuration. We believe that repeated propagation terminates for any configuration, but we do not prove this non-obvious[3] fact formally. Timely Dataflow also iterates propagation until all worklists of a worker become empty. This gives us additional empirical evidence that the iteration terminates on practical dataflows. Moreover, even if the iteration were to not terminate for some worker on some dataflow (both in Timely Dataflow and in our model), our combined protocol can faithfully capture this behavior by not executing the propagate action, but also not any other action involving the looping worker, thus retaining safety for the rest of the workers. Finally, any worker that has completed at least one propagation action is considered to be initialized (by setting its init flag to True).

The Init predicate and the Next relation give rise to the familiar specification of valid executions Spec s = now Init s ∧ alw (relates Next) s. Safety of the combined protocol can be described informally as follows: Every initialized worker w has some evidence for the existence of a timestamp t at location l at *any* worker w' in the frontier of its (i.e., w's) implications at all locations l' reachable from l. Formally, E.rec contains the timestamps that exist in the system:

definition Safe :: ('w, 'l, 't) *conf stream* ⇒ *bool* **where**
Safe c = ($\forall w\ l\ l'\ t\ s.$ init $c\ w \wedge$ E.rec (exch c) $\#_z\ (l, t) > 0 \wedge s \in$ path_summary $l\ l' \longrightarrow$
($\exists t' \in$ frontier (P.imp (prop $c\ w$) l'). $t' \leqslant t \oplus s$)

Our main formalized result is the statement that the above predicate is an invariant.

lemma *safe:* Spec $s \longrightarrow$ alw (now Safe) s

The proof proceeds by lifting (and then combining) the safety statements and some auxiliary invariants of the exchange protocol and the local propagation to the combined execution. The lifting step is feasible, because we included stutter steps in the modeling of these components. In particular, the projection of a valid execution to the exchange configurations results in a valid execution of the exchange protocol: the propagate step constitutes a stutter step for the exchange configuration. In contrast, the projection to the local propagation configuration does not result in a valid execution of the local propagation, but in an execution that takes steps according to the reflexive transitive closure of the local propagation's transition relation P.Next: the steps propagate and recv_upd can take an arbitrary number of local propagation steps (whereas other transitions stutter from the point of view of local propagation). Fortunately, safety properties are easy to lift to such "big-step" executions.

In the combined progress tracking protocol, safety guarantees that if a pointstamp is present at an operator's port, it is correctly reflected at every downstream port. In the WCC example, when deployed on two workers, each operator is instantiated twice, once on

[3] Because propagation must operate on a globally minimal timestamp and because loops in the dataflow graph have a non-zero summary, repeated propagation will eventually forever remove any timestamp from any worklist. However, it is not as obvious why it eventually will stop introducing larger and larger timestamps in worklists. The termination argument must rely on the fact that only timestamps that modify the frontier of the implications are ever added to worklists.

each worker. If a pointstamp (*b.3*, `(3,0)`) is present on port *b.3* of one of the instances of operator b, the frontier at *c.1* on all workers must contain a t such that $t \preceq (3,0)$. Due to the summary between *c.1* and *c.2*, frontiers at *c.2* and *b.1* must contain a t such that $t \preceq (3,1)$. As an example, this ensures that operator b waits for each of its instances to complete the first round propagation of all labels before it chooses the lowest label for the next round.

7 Discussion

We have presented an Isabelle/HOL formalization of Timely Dataflow's progress tracking protocol, including the verification of its safety. Compared to an earlier formalization by Abadi et al. [4], our protocol is both more general, which allows it to capture behaviors present in the implementations of Timely Dataflow and absent in Abadi et al.'s model, and more detailed in that it explicitly models the local propagation of progress information.

Our formalization spans about 7 000 lines of Isabelle definitions and proofs. These are roughly distributed as follows over the components we presented: basic properties of graphs and signed multisets (1 000), exchange protocol (3 100), local propagation (1 700), combined protocol (1 200). This is comparable in size to the TLA$^+$ Proof System formalization by Abadi et al., even though we formalized a significantly more detailed, complex, and realistic variant of the progress tracking protocol. Ground to this claim is the fact that we had actually started our formalization by porting significant parts of the TLA$^+$ Proof System formalization to Isabelle. We completed the proofs of their two main safety statement within one person-week in about 1 000 lines of Isabelle (not included above). Our use of Isabelle's library for linear temporal logic on streams (in particular, the coinductive predicate alw) allowed us to copy directly a vast majority of the TLA$^+$ definitions. Additionally, Isabelle's mature proof automation allowed us to apply a fairly mechanical porting process to many of the proofs. Most ported lemmas could be proved either directly by Sledgehammer [28] or by sketching an Isar [31] proof skeleton of the main proof steps and discharging most of the resulting subgoals with Sledgehammer.

In the subsequent development of the combined protocol, Isabelle's locales [5] were an important asset. By confining the exchange protocol and the local propagation each to their own local assumptions, we were able to develop them in parallel and in their full generality. Thus, we obtain formal models not only of the combined protocol itself but also of these two subsystems in a generality that goes beyond what is needed for the concrete combined instance. For example, although the combined protocol uses the could-result-in order, the exchange protocol works for any partial order on pointstamps. Moreover, the combined protocol always propagates until all worklists are empty, even though the local propagation's safety supports small-step propagation, resulting in a more fine-grained safety property via work_vacant.

In our formalization, we make extensive use of signed multisets [8]. The alternative (used in the TLA$^+$ Proof System formalization), would be to use integer-valued functions instead. The signed multiset type additionally captures a finite domain assumption, which it was convenient not to carry around explicitly and in particular simplified reasoning about summations. The expected downside of having separate types for function-like (*mset*) and set-like (*antichain*) objects was the need to insert explicit type conversions and to transfer properties across these conversions. Both complications were to some extent alleviated by Lifting and Transfer [18].

Progress tracking is only a small, albeit arguably the most intricate part of Timely Dataflow. Verifying its safety is an important first step towards our long-term goal of developing a verified, executable variant of Timely Dataflow and using it as a framework for the verification of efficient and scalable stream processing algorithms. More modest next steps are to prove the local propagation algorithm's termination and to make our formalization executable. We have made first steps towards the latter goal, by creating a functional, executable variant of the local propagation's transition relation [11]. This allowed us to compare our formalized propagation algorithm to the one implemented in Rust. We found that their input–output behavior coincides on all example dataflows accompanying the Rust implementation, confirming our model's faithfulness. We are working on including the exchange protocol in this comparative testing, which poses a challenge because of the protocol's distributed nature.

Acknowledgments We thank David Basin and Timothy Roscoe for supporting this work and Frank McSherry for providing valuable input on our formalization, e.g, by suggesting to consider the implied_frontier notion and to show that it is what local propagation computes. David Cock, Jon Howell, and Frank McSherry provided helpful feedback after reading early drafts of this paper. Dmitriy Traytel is supported by a Novo Nordisk Fonden Start Package Grant (NNF20OC0063462). Andrea Lattuada is supported by a Google PhD Fellowship.

References

1. Github: Timely dataflow. URL: `https://github.com/TimelyDataflow/timely-dataflow/`.
2. Materialize: Incrementally-updated materialized views. URL: `https://materialize.com`.
3. Martín Abadi and Michael Isard. Timely dataflow: A model. In Susanne Graf and Mahesh Viswanathan, editors, *FORTE 2015*, volume 9039 of *LNCS*, pages 131–145. Springer, 2015. `doi:10.1007/978-3-319-19195-9_9`.
4. Martín Abadi, Frank McSherry, Derek Gordon Murray, and Thomas L. Rodeheffer. Formal analysis of a distributed algorithm for tracking progress. In Dirk Beyer and Michele Boreale, editors, *FMOODS/FORTE 2013*, volume 7892 of *LNCS*, pages 5–19. Springer, 2013. `doi:10.1007/978-3-642-38592-6_2`.
5. Clemens Ballarin. Locales: A module system for mathematical theories. *J. Autom. Reason.*, 52(2):123–153, 2014. URL: `https://doi.org/10.1007/s10817-013-9284-7`.
6. Véronique Benzaken and Evelyne Contejean. A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In Assia Mahboubi and Magnus O. Myreen, editors, *CPP 2019*, pages 249–261. ACM, 2019. `doi:10.1145/3293880.3294107`.
7. Véronique Benzaken, Evelyne Contejean, Chantal Keller, and E. Martins. A Coq formalisation of SQL's execution engines. In Jeremy Avigad and Assia Mahboubi, editors, *ITP 2018*, volume 10895 of *LNCS*, pages 88–107. Springer, 2018. `doi:10.1007/978-3-319-94821-8_6`.
8. Jasmin Christian Blanchette, Mathias Fleury, and Dmitriy Traytel. Nested multisets, hereditary multisets, and syntactic ordinals in Isabelle/HOL. In Dale Miller, editor, *FSCD 2017*, volume 84 of *LIPIcs*, pages 11:1–11:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.FSCD.2017.11`.
9. Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the TLA+ proof system. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR 2010*, volume 6173 of *LNCS*, pages 142–148. Springer, 2010. `doi:10.1007/978-3-642-14203-1_12`.
10. Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *CIDR 2017*. www.cidrdb.org, 2017. URL: `http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf`.
11. Sára Decova. Modelling and verification of the Timely Dataflow progress tracking protocol. Master's thesis, ETH Zurich, Zurich, 2020. `doi:10.3929/ethz-b-000444762`.
12. Tomás Díaz, Federico Olmedo, and Éric Tanter. A mechanized formalization of GraphQL. In Jasmin Blanchette and Catalin Hritcu, editors, *CPP 2020*, pages 201–214. ACM, 2020. `doi:10.1145/3372885.3373822`.
13. Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.*, 1(OOPSLA):109:1–109:28, 2017. `doi:10.1145/3133933`.

14 Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *OSDI 2020*, pages 99–115. USENIX Association, 2020. URL: `https://www.usenix.org/conference/osdi20/presentation/hance`.

15 Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In Ethan L. Miller and Steven Hand, editors, *SOSP 2015*, pages 1–17. ACM, 2015. `doi:10.1145/2815400.2815428`.

16 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: session-type based reasoning in separation logic. *Proc. ACM Program. Lang.*, 4(POPL):6:1–6:30, 2020. `doi:10.1145/3371074`.

17 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. `doi:10.1145/359576.359585`.

18 Brian Huffman and Ondrej Kuncar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In Georges Gonthier and Michael Norrish, editors, *CPP 2013*, volume 8307 of *LNCS*, pages 131–146. Springer, 2013. `doi:10.1007/978-3-319-03545-1_9`.

19 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. `doi:10.1017/S0956796818000151`.

20 Leslie Lamport. Paxos made simple, fast, and Byzantine. In Alain Bui and Hacène Fouchal, editors, *OPODIS 2002*, volume 3 of *Studia Informatica Universalis*, pages 7–9. Suger, Saint-Denis, rue Catulienne, France, 2002.

21 Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In Rastislav Bodík and Rupak Majumdar, editors, *POPL 2016*, pages 357–370. ACM, 2016. `doi:10.1145/2837614.2837622`.

22 J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In Manuel V. Hermenegildo and Jens Palsberg, editors, *POPL 2010*, pages 237–248. ACM, 2010. `doi:10.1145/1706299.1706329`.

23 Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared arrangements: practical inter-query sharing for streaming dataflows. *Proc. VLDB Endow.*, 13(10):1793–1806, 2020. URL: `http://www.vldb.org/pvldb/vol13/p1793-mcsherry.pdf`.

24 Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR 2013*. www.cidrdb.org, 2013. URL: `http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf`.

25 Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In Michael Kaminsky and Mike Dahlin, editors, *SOSP 2013*, pages 439–455. ACM, 2013. `doi:10.1145/2517349.2522738`.

26 Derek Gordon Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martín Abadi. Incremental, iterative data processing with timely dataflow. *Commun. ACM*, 59(10):75–83, 2016. `doi:10.1145/2983551`.

27 Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In Garth Gibson and Nickolai Zeldovich, editors, *USENIX ATC 2014*, pages 305–319. USENIX Association, 2014. URL: `https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

28 Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *IWIL 2010*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010. URL: `https://easychair.org/publications/paper/wV`.

29 Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.*, 2(POPL):28:1–28:30, 2018. `doi:10.1145/3158116`.

30 Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David A. Basin. Igloo: soundly linking compositional refinement and separation logic for distributed system verification. *Proc. ACM Program. Lang.*, 4(OOPSLA):152:1–152:31, 2020. doi:10.1145/3428220.

31 Makarius Wenzel. Isabelle/Isar—A generic framework for human-readable proof documents. In Roman Matuszewski and Anna Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar, and Rhetoric*. Uniwersytet w Białymstoku, 2007.

32 James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In David Grove and Steve Blackburn, editors, *PLDI 2015*, pages 357–368. ACM, 2015. doi:10.1145/2737924.2737958.