

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...

Supplementary Material github.com/matthias-brun/progress-tracking-formalization

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 [22], incur significantly less intrinsic overhead [24].

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.



© Matthias Brun, Sára Decova, Andrea Lattuada, and Dmitriy Traytel;
licensed under Creative Commons License CC-BY 4.0

Leibniz International Proceedings in Informatics

LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

44 The progress tracking mechanism must be correct. Incorrect approximations of the
 45 frontiers can result in subtle concurrency errors, which may only appear under certain load
 46 and deployment circumstances. In this work, we formally model in Isabelle/HOL and prove
 47 the safety of the progress tracking protocol of *Timely Dataflow* [1, 24, 25] (Section 2), a
 48 time-aware dataflow programming model and a state-of-the-art streaming, data-parallel,
 49 distributed data processor.

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

55 This is an additional (worker-local) dimension in the specification when compared to
 56 well-known distributed coordination protocols, such as Paxos [19] and Raft [26], which focus
 57 on the interaction between symmetric communicating parties on different nodes. In contrast
 58 our environment model can be simpler, as progress tracking is designed to handle but not
 59 recover from fail-stop failures or unbounded pauses: upon crashes, unbounded stalls, or reset
 60 of a channel, the system stops without violating safety.¹

61 Abadi et al. [4] formalize and prove safety of the distributed exchange component of pro-
 62 gress tracking in the TLA⁺ Proof System. We present their *clocks protocol* through the lens
 63 of our Isabelle re-formalization (Section 3) and show that it subtly fails to capture behaviors
 64 supported by Timely Dataflow [24, 25]. We then significantly extend the formalized protocol
 65 (Section 4) to faithfully model Timely Dataflow’s modern reference implementation [1].

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

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

78 Related Work

79 **Data management systems verification** Timely Dataflow is a system that supports low-
 80 latency, high-throughput data-processing applications. Higher level libraries [22, 23] and
 81 SQL abstractions [2] built on Timely Dataflow support high performance incremental view
 82 maintenance for complex queries over large datasets. Verification and formal methods efforts
 83 in the data management and processing space have focused on SQL and query-language
 84 semantics [6, 10, 11] and on query runtimes in database management systems [7, 21].

85 **Distributed systems verification** Timely Dataflow is a distributed, concurrent system: our
 86 modeling and proof techniques are based on the widely accepted state machine model and
 87 refinement approach as used, e.g., in the TLA⁺ Proof System [9] and Ironfleet [14]. 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 [12, 13, 20] and providing tools for the implementation and verification of general distributed protocols [18] leveraging domain-specific languages [28, 30] and advanced type systems [15].

Model of Timely Dataflow Abadi and Isard [3] define abstractly the semantics of a Timely Dataflow programming model [24]. 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 [24]. 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 message queues—they act as communicating sequential processes [16]. 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.

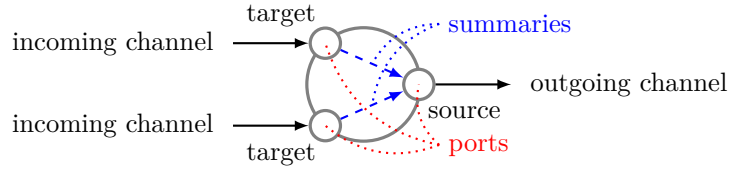


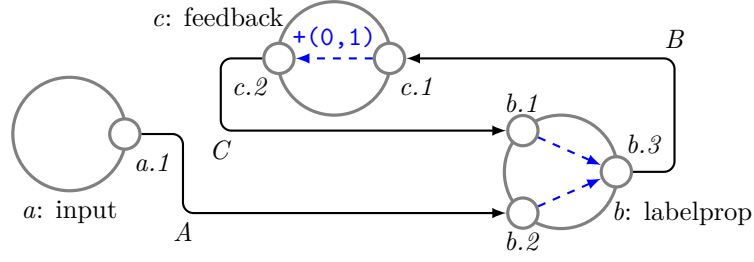
Figure 1 A Timely Dataflow operator.

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 $\exists t' \in F. t' \preceq t$.

XX:4 Verified Progress Tracking for Timely Dataflow



■ **Figure 2** A timely dataflow that computes weakly connected components.

123 **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.

128 ► **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 (a, b) 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 (a, b) by *a*.

134 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.

140 The input *a* starts with a pointstamp 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*. When it ceases sending data for a certain timestamp, e.g. $(0, 0)$, *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.

148 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 neighbours *b*, it sends (b, 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 (b, x) by *b*. 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*.

156 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

161 a fixed t_1 corresponding to the input version being considered. Operator b determines it has
 162 correctly labeled all connected components for a given t_1 when the frontier at $b.1$ does not
 163 contain a (t_1, t_2) such that $t_2 \preceq$ the graph's diameter. In practice, once operator b determines
 164 it has computed the output for a given t_1 , the operator would also send the output on an
 165 additional outgoing channel to deliver it to the user. After, operator b continues processing
 166 for further input versions, indicated by increasing t_1 , with timestamps $(t_1, 0)$, $(t_1, 1)$, etc. ◀

167 3 The Clocks Protocol

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

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

177 In Isabelle, we use the type variable $'w :: \text{finite}$ to represent workers. We assume that
 178 $'w$ belongs to the *finite* type class, which assures that $'w$'s universe is finite. Similarly, we
 179 model pointstamps abstractly by $'p :: \text{order}$. The *order* type class assumes the existence of a
 180 partial order $\leq :: 'p \Rightarrow 'p \Rightarrow \text{bool}$ (and the corresponding strict order $<$).

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

```
182 record ('w :: finite, 'p :: order) conf =
      rec :: 'p zmultiset
      msg :: 'w  $\Rightarrow$  'w  $\Rightarrow$  'p zmultiset list
      temp :: 'w  $\Rightarrow$  'p zmultiset
      glob :: 'w  $\Rightarrow$  'p zmultiset
```

183 Here, $\text{rec } c$ denotes the global multiset of pointstamps (or records) that are present in a
 184 system's configuration c . We use the type $'p \text{ zmultiset}$ of *signed multisets* [8]. An element
 185 $M :: 'p \text{ zmultiset}$ can be thought of as a function of type $'p \Rightarrow \text{int}$, which is non-zero only for
 186 finitely many values. (In contrast, an unsigned multiset $M :: 'p \text{ multiset}$ corresponds to a function
 187 of type $'p \Rightarrow \text{nat}$.) Signed multisets enjoy nice algebraic properties; in particular, they form a
 188 group. This significantly simplifies the reasoning about subtraction. However, $\text{rec } c$ will always
 189 store only non-negative pointstamp counts. The other components of a configuration c are

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

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

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

201 **definition** $\text{Init} :: ('w, 'p) \text{conf} \Rightarrow \text{bool}$ **where**
 202 $\text{Init } c = (\forall p. \text{rec } c \#_z p \geq 0) \wedge (\forall w \ w'. \text{msg } c \ w \ w' = []) \wedge$
 203 $(\forall w. \text{temp } c \ w = \{\#\}_z) \wedge (\forall w. \text{glob } c \ w = \text{rec } c)$

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

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

211 The configurations evolve via one of three actions:

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

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

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

216 Overall, the clocks protocol aims to establish that **glob** is a safe approximation for **rec**.
 217 Safe means here that no pointstamp in **rec** is less than any of **glob**'s minimal pointstamps. To
 218 achieve this property, the protocol imposes a restriction on which new pointstamps may be
 219 introduced in **rec** and which progress updates may be broadcast. This restriction is the *upright-*
 220 *ness* property: A signed multiset of pointstamps is upright if every positive entry is accompan-
 221 ied by a smaller negative entry. In other words, upright changes ensure that a pointstamp can
 222 only be introduced if simultaneously a smaller (supporting) pointstamp is removed. Formally:

223 **definition** $\text{supp} :: 'p \text{zmsset} \Rightarrow 'p \Rightarrow \text{bool}$ **where** $\text{supp } M \ p = (\exists p' < p. M \#_z p' < 0)$

224 **definition** $\text{upright} :: 'p \text{zmsset} \Rightarrow \text{bool}$ **where** $\text{upright } M = (\forall p. M \#_z p > 0 \longrightarrow \text{supp } M \ p)$

225 Abadi et al. [4] additionally require that the pointstamp p' in **supp**'s definition satisfies
 226 $\forall p'' \leq p'. M \#_z p'' \leq 0$. The two variants of **upright** are equivalent in our formalization
 227 because signed multisets are finite and thus minimal elements exist even without \leq being
 228 well-founded. The extra assumption on p' is occasionally useful in proofs.

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

236 Figure 3 defines the three protocol actions formally as transition relations between an old
 237 configuration c and a new configuration c' along with the definition of the overall transition
 238 relation **Next**. The three actions take further parameters as arguments, which we explain next.

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

definition $\text{perf_op} :: 'w \Rightarrow 'p \text{ mset} \Rightarrow 'p \text{ mset} \Rightarrow ('w, 'p) \text{ conf} \Rightarrow ('w, 'p) \text{ conf} \Rightarrow \text{bool}$ **where**
 $\text{perf_op } w \Delta_{\text{neg}} \Delta_{\text{pos}} c c' = \text{let } \Delta = \Delta_{\text{pos}} - \Delta_{\text{neg}} \text{ in } (\forall p. \Delta_{\text{neg}} \# p \leq \text{rec } c \#_z p) \wedge \text{upright } \Delta \wedge$
 $c' = c(\text{rec} = \text{rec } c + \Delta, \text{temp} = (\text{temp } c)(w := \text{temp } c \ w + \Delta))$

definition $\text{send_upd} :: 'w \Rightarrow 'p \text{ set} \Rightarrow ('w, 'p) \text{ conf} \Rightarrow ('w, 'p) \text{ conf} \Rightarrow \text{bool}$ **where**
 $\text{send_upd } w P c c' = \text{let } \gamma = \{\#p \in \#_z \text{temp } c \ w. p \in P\# \} \text{ in}$
 $\gamma \neq \{\# \}_z \wedge \text{upright } (\text{temp } c \ w - \gamma) \wedge$
 $c' = c(\text{msg} = (\text{msg } c)(w := \lambda w'. \text{msg } c \ w \ w' \cdot [\gamma]), \text{temp} = (\text{temp } c)(w := \text{temp } c \ w - \gamma))$

definition $\text{recv_upd} :: 'w \Rightarrow 'w \Rightarrow ('w, 'p) \text{ conf} \Rightarrow ('w, 'p) \text{ conf} \Rightarrow \text{bool}$ **where**
 $\text{recv_upd } w w' c c' = \text{msg } c \ w \ w' \neq [] \wedge$
 $c' = c(\text{msg} = (\text{msg } c)(w := (\text{msg } c \ w)(w' := \text{tl } (\text{msg } c \ w \ w'))),$
 $\text{glob} = (\text{glob } c)(w' := \text{glob } c \ w' + \text{hd } (\text{msg } c \ w \ w')))$

definition $\text{Next} :: ('w, 'p) \text{ conf} \Rightarrow ('w, 'p) \text{ conf} \Rightarrow \text{bool}$ **where**
 $\text{Next } c c' = (\exists w \Delta_{\text{neg}} \Delta_{\text{pos}}. \text{perf_op } w \Delta_{\text{neg}} \Delta_{\text{pos}} c c') \vee$
 $(\exists w P. \text{send_upd } w P c c') \vee (\exists w w'. \text{recv_upd } w w' c c')$

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

counts from Δ_{neg} must be bounded by the ones from $\text{rec } c$. Second, Δ must be upright, which among other things ensures that we will never introduce a pointstamp that is lower than any pointstamp in $\text{rec } c$. Once 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 enqueueing 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 $\text{msg } c \ w \ w'$, 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** $'a \text{ stream} = \text{Stream } 'a \ ('a \text{ stream})$. We can inspect a stream's head and tail using the functions $\text{shd} :: 'a \text{ stream} \Rightarrow 'a$ and $\text{stl} :: 'a \text{ stream} \Rightarrow 'a \text{ stream}$. Valid protocol executions satisfy the predicate Spec , i.e., they start in an initial configuration and all neighboring configurations are related by Next :

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

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: $\text{now } P \ s = P \ (\text{shd } s)$ and $\text{relates } R \ s = R \ (\text{shd } s) \ (\text{shd } (\text{stl } s))$. The coinductive operator alw resembles a temporal logic operator: $\text{alw } P \ s$ holds if P holds for all suffixes of s .

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

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:

XX:8 Verified Progress Tracking for Timely Dataflow

273 **definition** *vacant* :: $'p \text{ } z\text{mset} \Rightarrow 'p \Rightarrow \text{bool}$ **where** *vacant* $M \text{ } p = (\forall p' \leq p. M \#_z p' = 0)$

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

278 **definition** *Safe* :: $('w, 'p) \text{ } \text{conf } \text{stream} \Rightarrow \text{bool}$ **where**

$\text{Safe } s = (\forall w \text{ } p. \text{now } (\lambda c. \text{vacant } (\text{glob } c \text{ } w) \text{ } p) \text{ } s \longrightarrow \text{alw } (\text{now } (\lambda c. \text{vacant } (\text{rec } c) \text{ } p) \text{ } s))$

lemma *safe*: $\text{Spec } s \longrightarrow \text{alw } \text{Safe } s$

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

- 281 1. **rec** only contains positive entries
- 282 2. **rec** is the sum of any worker w 's **glob** and its *incoming information* $\text{info } c \text{ } w = \sum_{w'} (\text{temp } c \text{ } w' +$
283 $\sum_{M \in \text{set } (\text{msg } c \text{ } w' \text{ } w) } M)$, that is the sum of all workers' **temp** and all **msg** directed towards w
- 284 3. any worker w 's incoming information is upright

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

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

296 **definition** *Mono* :: $('w, 'p) \text{ } \text{conf} \Rightarrow ('w, 'p) \text{ } \text{conf} \Rightarrow \text{bool}$ **where**

$\text{Mono } c \text{ } c' = (\forall w \text{ } p. \text{vacant } (\text{glob } c \text{ } w) \text{ } p \longrightarrow \text{vacant } (\text{glob } c' \text{ } w) \text{ } p)$

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

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

307 **definition** *beta_upright* :: $'p \text{ } z\text{mset} \Rightarrow 'p \text{ } z\text{mset} \Rightarrow \text{bool}$ **where**

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

308 We do not describe in detail how beta uprightness helps with monotonicity, but the main
309 step is to establish the invariant that all messages M from **msg** are beta upright with respect
310 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 $(t_1, 0)$. This operator may be invoked multiple times, and perform this action repeatedly, until it finally 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 positive 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:

```
record ('w :: finite, 'p :: order) conf =
  caps :: 'w => 'p zmultiset
  data :: ('w × 'p) multiset
  msg :: 'w => 'w => 'p zmultiset list
  temp :: 'w => 'p zmultiset
  glob :: 'w => 'p zmultiset
```

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.

XX:10 Verified Progress Tracking for Timely Dataflow

definition `recv_cap` :: $'w \Rightarrow 'p \Rightarrow ('w, 'p) \text{ conf} \Rightarrow ('w, 'p) \text{ conf} \Rightarrow \text{bool}$ **where**
 $\text{recv_cap } w \ p \ c \ c' = (w, p) \in \# \text{ data } c \wedge$
 $c' = c(\text{caps} = (\text{caps } c)(w := \text{caps } c \ w + \{\#p\}_z), \text{data} = \text{data } c - \{\#(w, p)\}_z)$

definition `perf_op` :: $'w \Rightarrow 'p \text{ mset} \Rightarrow ('w \times 'p) \text{ mset} \Rightarrow 'p \text{ mset} \Rightarrow$
 $('w, 'p) \text{ conf} \Rightarrow ('w, 'p) \text{ conf} \Rightarrow \text{bool}$ **where**
 $\text{perf_op } w \ \Delta_{\text{neg}} \ \Delta_{\text{data}} \ \Delta_{\text{self}} \ c \ c' =$
 $(\Delta_{\text{data}} \neq \{\#\} \vee \Delta_{\text{self}} - \Delta_{\text{neg}} \neq \{\#\}_z) \wedge (\forall p. \Delta_{\text{neg}} \# p \leq \text{caps } c \ w \ \#_z \ p) \wedge$
 $(\forall (w', p) \in \# \Delta_{\text{data}}. \exists p' < p. \text{caps } c \ w \ \#_z \ p' > 0) \wedge$
 $(\forall p \in \# \Delta_{\text{self}}. \exists p' \leq p. \text{caps } c \ w \ \#_z \ p' > 0) \wedge$
 $c' = c(\text{caps} = (\text{caps } c)(w := \text{caps } c \ w + \Delta_{\text{self}} - \Delta_{\text{neg}}), \text{data} = \text{data } c + \Delta_{\text{data}},$
 $\text{temp} = (\text{temp } c)(w := \text{temp } c \ w + (\text{snd } \# \Delta_{\text{data}} + \Delta_{\text{self}} - \Delta_{\text{neg}})))$

definition `send_upd` :: $'w \Rightarrow 'p \text{ set} \Rightarrow ('w, 'p) \text{ conf} \Rightarrow ('w, 'p) \text{ conf} \Rightarrow \text{bool}$ **where**
 $\text{send_upd } w \ P \ c \ c' = \text{let } \gamma = \{\#p \in \#_z \text{ temp } c \ w. p \in P\} \text{ in}$
 $\gamma \neq \{\#\}_z \wedge \text{justified } (\text{caps } c \ w) (\text{temp } c \ w - \gamma) \wedge$
 $c' = c(\text{msg} = (\text{msg } c)(w := \lambda w'. \text{msg } c \ w \ w' \cdot [\gamma]), \text{temp} = (\text{temp } c)(w := \text{temp } c \ w - \gamma))$

definition `recv_upd` :: $'w \Rightarrow 'w \Rightarrow ('w, 'p) \text{ conf} \Rightarrow ('w, 'p) \text{ conf} \Rightarrow \text{bool}$ **where**
 $\text{recv_upd } w \ w' \ c \ c' = \text{msg } c \ w \ w' \neq [] \wedge$
 $c' = c(\text{msg} = (\text{msg } c)(w := (\text{msg } c \ w)(w' := \text{tl } (\text{msg } c \ w \ w'))),$
 $\text{glob} = (\text{glob } c)(w' := \text{glob } c \ w' + \text{hd } (\text{msg } c \ w \ w')))$

definition `Next` :: $('w, 'p) \text{ conf} \Rightarrow ('w, 'p) \text{ conf} \Rightarrow \text{bool}$ **where**
 $\text{Next } c \ c' = (c = c') \vee (\exists w \ p. \text{recv_cap } w \ p \ c \ c') \vee$
 $(\exists w \ \Delta_{\text{neg}} \ \Delta_{\text{data}} \ \Delta_{\text{self}}. \text{perf_op } w \ \Delta_{\text{neg}} \ \Delta_{\text{data}} \ \Delta_{\text{self}} \ c \ c') \vee$
 $(\exists w \ P. \text{send_upd } w \ P \ c \ c') \vee (\exists w \ w'. \text{recv_upd } w \ w' \ c \ c')$

■ **Figure 4** Transition relation of the exchange protocol

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

352 **definition** `rec` :: $('w, 'p) \text{ conf} \Rightarrow 'p \text{ zmset}$ **where** $\text{rec } c = (\sum_w \text{caps } c \ w) + \text{snd } \# \text{ data } c$

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

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

357 **definition** `Init` :: $('w, 'p) \text{ conf} \Rightarrow \text{bool}$ **where**
 $\text{Init } c = (\forall w \ p. \text{caps } c \ w \ \#_z \ p \geq 0) \wedge \text{data } c = \{\#\} \wedge$
 $(\forall w \ w'. \text{msg } c \ w \ w' = []) \wedge (\forall w. \text{temp } c \ w = \{\#\}_z) \wedge (\forall w. \text{glob } c \ w = \text{rec } c)$

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

362 The `perf_op` transition resembles its homonymous counterpart from the clocks protocol.
 363 Yet, the information flow is more fine grained. In particular, the transition is parameterized
 364 by a worker `w` and three multisets of pointstamps. As in the clocks protocol, the multiset
 365 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 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. $\forall w. \text{justified}(\text{rec } c)(\text{info } c \ w)$. 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 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 .

```

locale graph =
  fixes weights :: ('vtx :: finite)  $\Rightarrow$  'vtx  $\Rightarrow$  ('lbl :: {order, monoid_add}) antichain
  assumes (l :: 'lbl)  $\geq 0$  and (l1 :: 'lbl)  $\leq l_3 \longrightarrow l_2 \leq l_4 \longrightarrow l_1 + l_2 \leq l_3 + l_4$ 
  and weights l l = {}

locale dataflow = graph summary
  for summary :: ('l :: finite)  $\Rightarrow$  'l  $\Rightarrow$  ('sum :: {order, monoid_add}) antichain +
  fixes  $\oplus$  :: ('t :: order)  $\Rightarrow$  'sum  $\Rightarrow$  't
  assumes  $t \oplus 0 = t$  and  $(t \oplus s) \oplus s' = t \oplus (s + s')$  and  $t \leq t' \longrightarrow s \leq s' \longrightarrow t \oplus s \leq t' \oplus s'$ 
  and path l l xs  $\longrightarrow xs \neq [] \longrightarrow t < t \oplus (\sum xs)$ 

```

■ **Figure 5** Locales for graphs and dataflows

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

417 **definition** `justified_with` :: 'p `zmset` \Rightarrow 'p `zmset` \Rightarrow 'p `zmset` \Rightarrow bool **where**

$$\text{justified_with } C \ M \ N = (\forall p. M \#_z p > 0 \longrightarrow$$

$$(\exists p' < p. M \#_z p' < 0 \vee N \#_z p' < 0) \vee (\exists p' < p. C \#_z p' > 0) \vee (M + N) \#_z p < C \#_z p)$$

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

423 **lemma** `glob`: Spec $s \longrightarrow \text{alw } (\text{now } (\lambda c. \forall w \ w' \ p. \\$

$$(\exists M \in \text{set } (\text{msg } c \ w \ w'). p \in \#_z M) \longrightarrow (\exists p' \leq p. \text{glob } c \ w' \ \#_z p' > 0))) \ s$$

5 Locally Propagating Progress

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

436 Figure 5 details our graph and dataflow modeling, which uses locales [5] to capture our
 437 abstract assumptions on dataflows and timestamps. A locale lets us fix parameters (types
 438 and constants) and assume properties about them. In our setting, a weighted directed graph
 439 is given by a finite (class *finite*) type *'vtx* of vertices and a `weights` function that assigns each
 440 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 \leq . A weight is then an antichain of labels, that is a set of incomparable (with respect to \leq) labels, which we model as follows:

typedef (*'t :: order*) *antichain* = $\{A :: 't \text{ set. finite } A \wedge (\forall a \in A. \forall b \in A. a \not\leq b \wedge b \not\leq a)\}$

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 $l \in \text{weights } u \ v$ 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 \leq). 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 zmultiset* 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 \leq) 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.

definition *antichain_of* :: *'t set* \Rightarrow *'t set* **where** *antichain_of* *A* = $\{x \in A. \neg \exists y \in A. y < x\}$
lift_definition *frontier* :: *'t zmultiset* \Rightarrow *'t antichain* **is** $\lambda M. \text{antichain_of } \{t. M \#_z t > 0\}$

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 \oplus to define the implied frontier.

definition \oplus :: *'t zmultiset* \Rightarrow *'sum antichain* \Rightarrow *'t zmultiset* **where**

$$M \oplus A = \sum_{s \in A} (\lambda t. t \oplus s) \#_z M$$

definition *implied_frontier* :: (*'l* \Rightarrow *'t zmultiset*) \Rightarrow *'l* \Rightarrow *'t antichain* **where**

$$\text{implied_frontier } P \ l = \text{frontier } \left(\sum_{l'} (\text{pos}_z (P \ l') \oplus \text{path_summary } l' \ l) \right)$$

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 $\text{pos}_z M$ for the signed multiset of *M*'s positive entries.

XX:14 Verified Progress Tracking for Timely Dataflow

definition `change_multiplicity` :: $'l \Rightarrow 't \Rightarrow \text{int} \Rightarrow ('l, 't) \text{ conf} \Rightarrow ('l, 't) \text{ conf} \Rightarrow \text{bool}$ **where**
`change_multiplicity` $l\ t\ n\ c' = n \neq 0 \wedge (\exists t' \in \text{frontier}(\text{implications } c\ l). t' \leq t) \wedge$
 $c' = c(\text{pts} = (\text{pts } c)(l := \text{pts } c\ l + \text{replicate } n\ t),$
 $\text{work} = (\text{work } c)(l := \text{work } c\ l + \text{frontier}(\text{pts } c'\ l) - \text{frontier}(\text{pts } c\ l)))$

definition `propagate` :: $'l \Rightarrow 't \Rightarrow ('l, 't) \text{ conf} \Rightarrow ('l, 't) \text{ conf} \Rightarrow \text{bool}$ **where**
`propagate` $l\ t\ c' = t \in \#_z \text{work } c\ l \wedge (\forall l'. \forall t' \in \#_z \text{work } c\ l'. \neg t' < t) \wedge$
 $c' = c(\text{imp} = (\text{imp } c)(l := \text{imp } c\ l + \text{replicate}(\text{work } c\ l\ \#_z\ t)\ t,$
 $\text{work} = \lambda l'. \text{if } l = l' \text{ then } \{\#t' \in \#_z \text{work } c\ l. t' \neq t\}$
 $\text{else } \text{work } c\ l' + ((\text{frontier}(\text{imp } c'\ l) - \text{frontier}(\text{imp } c\ l)) \oplus \text{summary } l\ l')))$

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

■ **Figure 6** Transition relation of the local progress propagation

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

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

490 **record** $('l :: \text{finite}, 't :: \{\text{monoid_add}, \text{order}\}) \text{ conf} =$
 $\text{pts} :: 'l \Rightarrow 't \text{ zmultiset}$
 $\text{imp} :: 'l \Rightarrow 't \text{ zmultiset}$
 $\text{work} :: 'l \Rightarrow 't \text{ zmultiset}$

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

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

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

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

506 The second action, `propagate`, applies the information for the timestamp t stored in the
 507 worklist at a given location l , to the location's implications (thus potentially enabling the first
 508 action). It also updates the worklists at the location's immediate successors in the dataflow
 509 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: $\text{Spec } s = \text{now Init } s \wedge \text{alw } (\text{relates Next}) s$.

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 $\text{work_vacant} :: ('l, 't) \text{ conf} \Rightarrow 't \Rightarrow \text{bool}$ **where**

$\text{work_vacant } c \ t = (\forall l' \ s \ t'. t' \in \#_z \text{work } c \ l \longrightarrow s \in \text{path_summary } l \ l' \longrightarrow t' \oplus s \not\leq t)$

definition $\text{Safe} :: ('l, 't) \text{ conf stream} \Rightarrow \text{bool}$ **where**

$\text{Safe } c = (\forall l' \ t \ s. \text{pts } c \ l \ \#_z \ t > 0 \wedge s \in \text{path_summary } l \ l' \wedge \text{work_vacant } c \ (t \oplus s) \longrightarrow (\exists l' \in \text{frontier } (\text{imp } c \ l'). t' \leq t \oplus s))$

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

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 $\text{summary } l' \ l$).

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*: $\text{Spec } s \longrightarrow \text{alw } (\text{now } (\lambda c. \text{work_vacant } c \ t \longrightarrow$

$(\forall l. t \in \text{frontier } (\text{imp } c \ l) \longleftrightarrow t \in \text{implied_frontier } (\text{pts } c \ l))) s$

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.

record $('w :: \text{finite}, 'l :: \text{finite}, 't :: \{\text{monoid_add}, \text{order}\}) \text{ conf} =$

$\text{exch} :: ('w, 'l \times 't) \text{ E.conf}$

$\text{prop} :: 'w \Rightarrow ('l, 't) \text{ P.conf}$

$\text{init} :: 'w \Rightarrow \text{bool}$

XX:16 Verified Progress Tracking for Timely Dataflow

definition `recv_cap` :: $'w \Rightarrow 'l \times 't \Rightarrow ('w, 'l, 't) \text{ conf} \Rightarrow ('w, 'l, 't) \text{ conf} \Rightarrow \text{bool}$ **where**
`recv_cap` $w \ p \ c \ c' = \text{E.recv_cap } w \ p \ (\text{exch } c) \ (\text{exch } c') \wedge \text{prop } c' = \text{prop } c \wedge \text{init } c' = \text{init } c$

definition `perf_op` :: $'w \Rightarrow ('l \times 't) \text{ mset} \Rightarrow ('w \times ('l \times 't)) \text{ mset} \Rightarrow ('l \times 't) \text{ mset} \Rightarrow$
 $('w, 'l, 't) \text{ conf} \Rightarrow ('w, 'l, 't) \text{ conf} \Rightarrow \text{bool}$ **where**
`perf_op` $w \ \Delta_{\text{neg}} \ \Delta_{\text{data}} \ \Delta_{\text{self}} \ c \ c' = \text{E.perf_op } w \ \Delta_{\text{neg}} \ \Delta_{\text{data}} \ \Delta_{\text{self}} \ (\text{exch } c) \ (\text{exch } c') \wedge$
 $\text{prop } c' = \text{prop } c \wedge \text{init } c' = \text{init } c$

definition `send_upd` :: $'w \Rightarrow ('l \times 't) \text{ set} \Rightarrow ('w, 'l, 't) \text{ conf} \Rightarrow ('w, 'l, 't) \text{ conf} \Rightarrow \text{bool}$ **where**
`send_upd` $w \ P \ c \ c' = \text{E.send_upd } (\text{exch } c) \ (\text{exch } c') \ w \ P \wedge \text{prop } c' = \text{prop } c \wedge \text{init } c' = \text{init } c$

definition `cm_all` :: $('l, 't) \ P.\text{conf} \Rightarrow ('l \times 't) \text{ zmsset} \Rightarrow ('l, 't) \ P.\text{conf}$ **where**
`cm_all` $c \ \Delta = \text{Set.fold } (\lambda(l, t) \ c. \text{SOME } c'. \ P.\text{change_multiplicity } c \ c' \ l \ t \ (\Delta \#_z (l, t))) \ c$
 $\{(l, t). (l, t) \in \#_z \Delta\}$

definition `recv_upd` :: $'w \Rightarrow 'w \Rightarrow ('w, 'l, 't) \text{ conf} \Rightarrow ('w, 'l, 't) \text{ conf} \Rightarrow \text{bool}$ **where**
`recv_upd` $w \ w' \ c \ c' = \text{init } c \ w' \wedge \text{E.recv_upd } w \ t \ (\text{exch } c) \ (\text{exch } c') \wedge$
 $\text{prop } c' = (\text{prop } c)(w' := \text{cm_all } (\text{prop } c \ w') \ (\text{hd } (\text{E.msg } (\text{exch } c)))) \wedge \text{init } c' = \text{init } c$

definition `propagate` :: $'w \Rightarrow ('w, 'l, 't) \text{ conf} \Rightarrow ('w, 'l, 't) \text{ conf} \Rightarrow \text{bool}$ **where**
`propagate` $w \ c \ c' = \text{exch } c' = \text{exch } c \wedge \text{init } c' = (\text{init } c)(w := \text{True}) \wedge$
 $(\text{Some } \circ \text{prop } c') = (\text{Some } \circ \text{prop } c)(w := \text{while_option}$
 $(\lambda c. \exists l. P.\text{work } c \ l \neq \{\#\}_z) \ (\lambda c. \text{SOME } c'. \exists t. P.\text{propagate } l \ t \ c \ c') \ (\text{prop } c \ w))$

definition `Next` :: $('w, 'l, 't) \text{ conf} \Rightarrow ('w, 'l, 't) \text{ conf} \Rightarrow \text{bool}$ **where**
`Next` $c \ c' = (\exists w \ p. \text{recv_cap } w \ p \ c \ c') \vee$
 $(\exists w \ \Delta_{\text{neg}} \ \Delta_{\text{data}} \ \Delta_{\text{self}}. \text{perf_op } w \ \Delta_{\text{neg}} \ \Delta_{\text{data}} \ \Delta_{\text{self}} \ c \ c') \vee$
 $(\exists w \ P. \text{send_upd } w \ P \ c \ c') \vee (\exists w \ w'. \text{recv_upd } w \ w' \ c \ c') \vee (\exists w. \text{propagate } w \ c \ c')$

■ **Figure 7** Transition relation of the combined progress tracker

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

551 **definition** \leq_{cri} **where** $(l, t) \leq_{\text{cri}} (l', t') = (\exists s \in \text{path_summary } l \ l'. t \oplus s \leq t')$

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

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

561 **definition** `Init` :: $('w, 'l, 't) \text{ conf} \Rightarrow \text{bool}$ **where**
`Init` $c = (\forall w. \text{init } c \ w = \text{False}) \wedge \text{E.Init } (\text{exch } c) \wedge (\forall w. P.\text{Init } (\text{prop } c \ w)) \wedge$
 $(\forall w \ l \ t. P.\text{pts } (\text{prop } c \ w) \ l \ \#_z \ t = \text{E.glob } (\text{exch } c) \ w \ \#_z (l, t))$

562 Figure 7 shows the combined protocol's transition relation `Next`. Most actions have
 563 identical names as the exchange protocol's actions and they mostly perform the correspond-
 564 ing actions on the exchange part of the configuration. In addition, the `recv_upd` action
 565 also performs several `change_multiplicity` local propagation actions: the receiver updates
 566 the state of its local propagation configuration for all received timestamp updates. The
 567 action `propagate` does not have a counterpart in the exchange protocol. It iterates, using
 568 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 $s' = c \ (\dots (c \ s))$ 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-obvious³ 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 $\text{Spec } s = \text{now Init } s \wedge \text{alw } (\text{relates Next}) \ s$. 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 :: $(w, l, t) \text{ conf stream} \Rightarrow \text{bool}$ **where**

$$\text{Safe } c = (\forall w \ l' \ t \ s. \text{init } c \ w \wedge \text{E.rec } (\text{exch } c) \#_z (l, t) > 0 \wedge s \in \text{path_summary } l \ l' \longrightarrow (\exists t' \in \text{frontier } (\text{P.imp } (\text{prop } c \ w) \ l'). t' \leq t \oplus s))$$

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

lemma safe: $\text{Spec } s \longrightarrow \text{alw } (\text{now Safe}) \ s$

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 each worker. If a pointstamp at $(3, 0)$ 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.

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.

609 7 Discussion

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

615 Our formalization spans about 7 000 lines of Isabelle definitions and proofs. These are
 616 roughly distributed as follows over the components we presented: basic properties of **graphs**
 617 and signed multisets (1 000), exchange protocol (3 100), local propagation (1 700), combined
 618 protocol (1 200). This is comparable in size to the TLA^+ Proof System formalization by Abadi
 619 et al., even though we formalized a significantly more detailed, complex, and realistic variant of
 620 the progress tracking protocol. Ground to this claim is the fact that we had actually started our
 621 formalization by porting significant parts of the TLA^+ Proof System formalization to Isabelle.
 622 We completed the proofs of their two main safety statement within one person-week in about
 623 1 000 lines of Isabelle (not included above). Our use of Isabelle’s library for linear temporal
 624 logic on streams (in particular, the coinductive predicate **alw**) allowed us to copy directly a vast
 625 majority of the TLA^+ definitions. Additionally, Isabelle’s mature proof automation allowed us
 626 to apply a fairly mechanical porting process to many of the proofs. Most ported lemmas could
 627 be proved either directly by Sledgehammer [27] or by sketching an Isar [29] proof skeleton
 628 of the main proof steps and discharging most of the resulting subgoals with Sledgehammer.

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

638 In our formalization, we make extensive use of signed multisets [8]. The alternative (used in
 639 the TLA^+ Proof System formalization), would be to use integer-valued functions instead. The
 640 signed multiset type additionally captures a finite domain assumption, which it was convenient
 641 not to carry around explicitly and in particular simplified reasoning about summations. The
 642 expected downside of having separate types for function-like (*mset*) and set-like (*antichain*)
 643 objects was the need to insert explicit type conversions and to transfer properties across these
 644 conversions. Both complications were to some extent alleviated by Lifting and Transfer [17].

645 Progress tracking is only a small, albeit arguably the most intricate part of Timely Data-
 646 flow. Verifying its safety is an important first step towards our long-term goal of developing a
 647 verified, executable variant of Timely Dataflow and using it as a framework for the verification
 648 of efficient and scalable stream processing algorithms. More modest next steps are to prove
 649 the local propagation algorithm’s termination and to make our formalization executable. We
 650 have made first steps towards the latter goal, by creating a functional, executable variant
 651 of the local propagation’s transition relation. This allowed us to compare our formalized
 652 propagation algorithm to the one implemented in Rust. We found that their input–output
 653 behavior coincides on all example dataflows accompanying the Rust implementation, con-
 654 firming our model’s faithfulness. We are working on including the exchange protocol in this
 655 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. 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, *FMODS/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 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.
- 12 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.
- 13 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>.
- 14 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.
- 15 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.

- 707 16 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
708 doi:10.1145/359576.359585.
- 709 17 Brian Huffman and Ondrej Kuncar. Lifting and Transfer: A modular design for quotients in
710 Isabelle/HOL. In Georges Gonthier and Michael Norrish, editors, *CPP 2013*, volume 8307 of
711 *LNCS*, pages 131–146. Springer, 2013. doi:10.1007/978-3-319-03545-1_9.
- 712 18 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek
713 Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation
714 logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.
- 715 19 Leslie Lamport. Paxos made simple, fast, and Byzantine. In Alain Bui and Hacène Fouchal,
716 editors, *OPODIS 2002*, volume 3 of *Studia Informatica Universalis*, pages 7–9. Suger, Saint-
717 Denis, rue Catulienne, France, 2002.
- 718 20 Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent
719 distributed key-value stores. In Rastislav Bodík and Rupak Majumdar, editors, *POPL 2016*,
720 pages 357–370. ACM, 2016. doi:10.1145/2837614.2837622.
- 721 21 J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified
722 relational database management system. In Manuel V. Hermenegildo and Jens Palsberg,
723 editors, *POPL 2010*, pages 237–248. ACM, 2010. doi:10.1145/1706299.1706329.
- 724 22 Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared ar-
725 rangements: practical inter-query sharing for streaming dataflows. *Proc. VLDB Endow.*,
726 13(10):1793–1806, 2020. URL: <http://www.vldb.org/pvldb/vol13/p1793-mcsherry.pdf>.
- 727 23 Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential
728 dataflow. In *CIDR 2013*. www.cidrdb.org, 2013. URL: [http://cidrdb.org/cidr2013/Papers/](http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf)
729 [CIDR13_Paper111.pdf](http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf).
- 730 24 Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and
731 Martín Abadi. Naiad: a timely dataflow system. In Michael Kaminsky and Mike Dahlin,
732 editors, *SOSP 2013*, pages 439–455. ACM, 2013. doi:10.1145/2517349.2522738.
- 733 25 Derek Gordon Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and
734 Martín Abadi. Incremental, iterative data processing with timely dataflow. *Commun. ACM*,
735 59(10):75–83, 2016. doi:10.1145/2983551.
- 736 26 Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In
737 Garth Gibson and Nikolai Zeldovich, editors, *USENIX ATC 2014*, pages 305–319. USENIX As-
738 sociation, 2014. URL: [https://www.usenix.org/conference/atc14/technical-sessions/](https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro)
739 [presentation/ongaro](https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro).
- 740 27 Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with
741 Sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff
742 Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *IWIL 2010*, volume 2 of *EPiC Series*
743 *in Computing*, pages 1–11. EasyChair, 2010. URL: [https://easychair.org/publications/](https://easychair.org/publications/paper/wV)
744 [paper/wV](https://easychair.org/publications/paper/wV).
- 745 28 Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed
746 protocols. *Proc. ACM Program. Lang.*, 2(POPL):28:1–28:30, 2018. doi:10.1145/3158116.
- 747 29 Makarius Wenzel. Isabelle/Isar—A generic framework for human-readable proof documents.
748 In Roman Matuszewski and Anna Zalewska, editors, *From Insight to Proof: Festschrift in*
749 *Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar, and Rhetoric*.
750 Uniwersytet w Białymstoku, 2007.
- 751 30 James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D.
752 Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying
753 distributed systems. In David Grove and Steve Blackburn, editors, *PLDI 2015*, pages 357–368.
754 ACM, 2015. doi:10.1145/2737924.2737958.