Timely Dataflow: A Model

Martín Abadi, Michael Isard

▶ **To cite this version:**

Martín Abadi, Michael Isard. Timely Dataflow: A Model. 35th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE), Jun 2015, Grenoble, France. pp.131-145, 10.1007/978-3-319-19195-9_9 . hal-01767326

HAL Id: hal-01767326
https://inria.hal.science/hal-01767326v1

Submitted on 16 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Timely Dataflow: A Model

Martín Abadi[1,2] and Michael Isard[3]

[1] Google, Mountain View, California, USA
[2] University of California, Santa Cruz, California, USA
[3] Microsoft Research, Mountain View, California, USA*

Abstract. This paper studies timely dataflow, a model for data-parallel computing in which each communication event is associated with a virtual time. It defines and investigates the could-result-in relation which is central to this model, then the semantics of timely dataflow graphs.

1 Introduction

Timely dataflow is a model of data-parallel computation that extends traditional dataflow (e.g., [10]) by associating each communication event with a virtual time [12]. Virtual times need not be linearly ordered, nor correspond to the order in which events are processed. As in the Time Warp mechanism [7], virtual times serve to differentiate between data in different phases or aspects of a computation, for example data associated with different batches of inputs and different loop iterations. Thus, an implementation may overlap, but still distinguish, work that corresponds to multiple logical parts of a computation.

In this model, each node in a dataflow graph can request to be notified when it has received all messages for a given virtual time. The facilities for asynchronous processing and completion notifications imply that, even within a single program, some components can function in batch mode (queuing inputs and delaying processing until an appropriate notification) and others in streaming mode (processing inputs as they arrive). For example, an application may process a stream of GPS readings; as these readings arrive, the application may update a map and, after each batch of readings, recompute shortest paths between landmarks.

The Naiad system [12] is the origin and an embodiment of timely dataflow. Naiad aspires to serve as a coherent platform for data-parallel applications, offering both high throughput and low latency. Timely dataflow is crucial to this goal. Naiad contrasts with other systems that focus on narrower domains (e.g., graph problems) or on particular classes of programs (e.g., without loops).

The development and presentation of timely dataflow in the context of Naiad was fairly precise but informal. Only one of its critical components (a distributed algorithm that keeps track of virtual times for which there may remain work) was rigorously specified and verified [4]. Moreover, in the context of Naiad, definitions focus on particular structures of dataflow graphs and particular types

* Most of this work was done at Microsoft Research.

of nodes. Specifically, Naiad supports iterative computations, with loops that include special nodes for ingress, feedback, and egress, and with a set of virtual times that includes coordinates for input epochs and loop counters.

The goal of this paper is to provide a general, rigorous definition of timely dataflow. We allow arbitrary graph structures, partial orders of virtual times, and stateful local computations at each of the nodes. The local computations are deterministic (only for simplicity); non-determinism is introduced by the ordering of events. We specify the semantics of timely dataflow graphs using a linear-time temporal logic. In this setting, we explore some of the fundamental concepts and properties of the model. In particular, we study the could-result-in relation, which drives completion notifications; for instance, we investigate how it applies to recursive dataflow computations, which are beyond Naiad's present scope. The semantics serves as the basis for rigorous proofs, as we demonstrate with an example application. We are finding the semantics valuable in other, more substantial applications. Specifically, the results of this paper have already been useful to us in our work on information-flow security properties [1] and on fault-tolerance [2]. Our rather elementary formulation of the semantics amply suffices for these present purposes; we leave algebraic or categorical presentations (see, e.g., [6]) for further work.

The next section defines dataflow graphs and other basic notions. Section 3 concerns the could-result-in relation. Section 4 describes the semantics of graphs, and Section 5 applies it. Section 6 concludes. Because of space constraints, proofs are omitted.

2 Dataflow Graphs, Messages, and Times

As is typical in dataflow models, we specify computations as directed graphs, with distinguished input and output edges. The graphs may contain cycles. During execution, stateful nodes send and receive timestamped messages, and in addition may request and receive notifications that they have received all messages with a certain timestamp. This section defines the graphs and the behavior of individual nodes; later sections cover more global aspects of the semantics.

We write $\emptyset$ both for the empty sequence and for the empty set. We write $\langle\!\langle m_0, m_1, \ldots \rangle\!\rangle$ for the sequence (finite or infinite) that consists of m_0, m_1, ... We use "$\cdot$" for sequence concatenation and also for appending elements to sequences, for example writing $m \cdot u$ instead of $\langle\!\langle m \rangle\!\rangle \cdot u$, where u is a sequence and m an element. A mapping f on elements is extended to a mapping on sequences by letting $f(\langle\!\langle m_0, m_1, m_2, \ldots \rangle\!\rangle) = \langle\!\langle f(m_0), f(m_1), \ldots \rangle\!\rangle$, and to a mapping on sets by letting $f(S) = \{f(s) : s \in S\}$. When A is a set, we write $\mathcal{P}(A)$ for its powerset, and A^* and A^ω, respectively, for the sets of finite and infinite sequences of elements of A. When f is a function with a domain that includes A, we write $f{\restriction}A$ for the restriction of f to A. When B is also a set, we write $\Pi_{x \in A}.B$ for the set of functions that map each $x \in A$ to an element of B; if A is a finite set $\{a_1, \ldots, a_k\}$ and $b_1, \ldots, b_k$ are elements of B, we write such a function $\langle a_1 \mapsto b_1, \ldots, a_k \mapsto b_k \rangle$.

2.1 Basics of Graphs, Messages, and Times

We assume a set of *messages* M, a partial order of *times* $(T, \leq)$, and a time $\mathit{time}(m) \in T$ for each $m \in M$. We also assume a finite set of *nodes (processors)* P, and a set of *local states* Σ_{Loc} for them. Finally, we assume a set of *edges (channels)*, partitioned into *input* edges I, *internal* edges E, and *output* edges O. Edges have *sources* and *destinations* (not always both): for each $i \in I$, $\mathit{dst}(i) \in P$, and $\mathit{src}(i)$ is undefined; for each $e \in E$, $\mathit{src}(e), \mathit{dst}(e) \in P$, and we require that they are distinct; for each $o \in O$, $\mathit{src}(o) \in P$, and $\mathit{dst}(o)$ is undefined.

Input edges are not essential for computations getting started, because nodes can initially create data in response to notifications. We include input edges as a convenience, and because they can serve for connecting graphs.

We allow (but do not require) the set of times to be the disjoint union of multiple "time domains". For example, a node may receive inputs tagged with GMT times, and produce outputs tagged with GMT times, PST times, or perhaps with sequence numbers, and may even send outputs in different time domains along different edges. In Naiad, the nodes for loop ingress and egress, respectively, add and remove time coordinates that represent loop counters. Accordingly, we do not assume, for example, that it is always immediately meaningful to compare the times of inputs and outputs.

2.2 Processor Behavior

Timely dataflow supports stateful computations in which each node maintains local state. For each node p, a subset $\mathit{Initial}(p)$ of $\Sigma_{\mathrm{Loc}} \times \mathcal{P}(T)$ describes the possible initial states and initial notification requests for p. A *local history* for p is a finite sequence of the form $\langle\!\langle (s, N), x_1, \ldots, x_n \rangle\!\rangle$ where $(s, N) \in \mathit{Initial}(p)$, $n \geq 0$, and each x_i is either a pair (d, m) where $m \in M$ and $d \in I \cup E$ with $\mathit{dst}(d) = p$, or a time $t \in T$. In this context, we call a pair (d, m) or a time t an *event*. Thus, a local history records the order in which a node consumes events; it also determines what the node does in response to these events, via the function g_1 introduced below. We write $\mathit{Histories}(p)$ for the set of local histories of node p.

For each node p, the function $g_1(p)$ maps $\Sigma_{\mathrm{Loc}} \times (T \cup (\{d \in I \cup E \mid \mathit{dst}(d) = p\} \times M))$ to $\Sigma_{\mathrm{Loc}} \times \mathcal{P}(T) \times (\Pi_{\{d \in E \cup O \mid \mathit{src}(d) = p\}}.M^*)$. Intuitively g_1 describes one step of computation by one node:

- $g_1(p)(s, t) = (s', \{t_1, \ldots, t_n\}, \langle e_1 \mapsto \mu_1, \ldots, e_k \mapsto \mu_k \rangle)$ means that, in response to a notification for time t and at a state s, the node p can move to state s', request notifications for times $t_1, \ldots, t_n$, and add message sequences $\mu_1, \ldots, \mu_k$ on outgoing edges $e_1, \ldots, e_k$, respectively.
- $g_1(p)(s, (d, m)) = (s', \{t_1, \ldots, t_n\}, \langle e_1 \mapsto \mu_1, \ldots, e_k \mapsto \mu_k \rangle)$ means that, in response to a message m on incoming edge d and at a state s, the node p can move to state s', request notifications for times $t_1, \ldots, t_n$, and add message sequences $\mu_1, \ldots, \mu_k$ on outgoing edges $e_1, \ldots, e_k$, respectively.

We could easily restrict these definitions so that a message for time t cannot appear in a history to the right of a notification for time t, and so that notifications appear only in response to notification requests. However, such restrictions do not seem necessary; each node can enforce them.

We extend the function g_1 to a function g that applies to local histories. For each node p, $g(p)$ maps $Histories(p)$ to $\Sigma_{\mathrm{Loc}} \times \mathcal{P}(T) \times (\Pi_{\{d \in E \cup O \mid src(d)=p\}} M^*)$, and is defined inductively by:

- $g(p)(\langle\!\langle (s, N) \rangle\!\rangle) = (s, N, \langle e_1 \mapsto \emptyset, \ldots, e_k \mapsto \emptyset \rangle)$
- If $g(p)(h) = (s', N, \langle e_1 \mapsto \mu_1, \ldots, e_k \mapsto \mu_k \rangle)$, $h' = h \cdot t$, and $g_1(p)(s', t) = (s", N', \langle e_1 \mapsto \mu'_1, \ldots, e_k \mapsto \mu'_k \rangle)$, then

$$g(p)(h') = (s", N - \{t\} \cup N', \langle e_1 \mapsto \mu_1 \cdot \mu'_1, \ldots, e_k \mapsto \mu_k \cdot \mu'_k \rangle)$$

- If $g(p)(h) = (s', N, \langle e_1 \mapsto \mu_1, \ldots, e_k \mapsto \mu_k \rangle)$, $h' = h \cdot (d, m)$, and $g_1(p)(s', (d, m)) = (s", N', \langle e_1 \mapsto \mu'_1, \ldots, e_k \mapsto \mu'_k \rangle)$, then

$$g(p)(h') = (s", N \cup N', \langle e_1 \mapsto \mu_1 \cdot \mu'_1, \ldots, e_k \mapsto \mu_k \cdot \mu'_k \rangle)$$

Given a triple $(s, N, \langle e_1 \mapsto \mu_1, \ldots, e_k \mapsto \mu_k \rangle)$, perhaps obtained via one of these functions, we write: $\Pi_{\mathrm{Loc}}(s, N, \langle e_1 \mapsto \mu_1, \ldots, e_k \mapsto \mu_k \rangle)$ for s, $\Pi_{\mathrm{NR}}(s, N, \langle e_1 \mapsto \mu_1, \ldots, e_k \mapsto \mu_k \rangle)$ for N, and $\Pi_{e_i}(s, N, \langle e_1 \mapsto \mu_1, \ldots, e_k \mapsto \mu_k \rangle)$ for μ_i.

In this model, each node can consume and produce multiple events in one atomic action. For example, a node may simultaneously dequeue an input message and produce two output messages on each of two distinct edges. Alternative models could be more asynchronous; in our example, the node would first dequeue the input message, and after some delay produce the two output messages one after the other. Fortunately, such an asynchronous model can be seen as a special case of ours: in our model, asynchronous behavior can be produced by buffering (see, e.g., [14]). We say that $p \in P$ is a buffer node if there exist exactly one $e_1 \in I \cup E$ such that $dst(e_1) = p$ and exactly one $e_2 \in E \cup O$ such that $src(e_2) = p$, and $g_1(p)(s, t) = (s, \emptyset, \langle e_2 \mapsto \emptyset \rangle)$ and $g_1(p)(s, (e_1, m)) = (s, \emptyset, \langle e_2 \mapsto \langle\!\langle m \rangle\!\rangle \rangle)$. Such a node p is simply a relay between queues. (The term "buffer" comes from the literature.) In order to simulate a more asynchronous semantics, we could require that every non-buffer node has its output edges going into buffer nodes. However, we do not need to impose this constraint.

3 Pointstamps and the Could-result-in Relation

As indicated in the Introduction, each node can request to be notified when it has received all messages for a given virtual time. Furthermore, "under the covers", an implementation may benefit from knowing that a virtual time is complete in order to reclaim associated resources. Thus, the notion of completion of virtual times is central to timely dataflow and to its practical realization. Reasoning about completion is based on the *could-result-in* relation on *pointstamps*. In this section we define this relation and establish some of its properties.

3.1 Defining Could-result-in

A pointstamp is a pair (x, t) of a location x (node or edge) in a graph and a time t. Thus, the set of pointstamps is $((I \cup E \cup O) \cup P) \times T$. We say that pointstamp (x, t) could-result-in pointstamp (x', t'), and write $(x, t) \rightsquigarrow (x', t')$, if a message or notification at location x and time t may lead to a message or notification at location x' and time t'. We define $\rightsquigarrow$ via an auxiliary relation $\rightsquigarrow^1$ that reflects one step of computation.

Definition 1. *$(p, t) \rightsquigarrow^1 (d, t')$ if and only if $src(d) = p$ and there exist a history h for p and a state s such that*

$$g(p)(h) = (s, \ldots)$$

and an event x such that either $x = t$ or $x = (e, m)$ for some e and m such that $t = time(m)$, and

$$g_1(p)(s, x) = (\ldots, \langle \ldots d \mapsto \mu \ldots \rangle)$$

where some element of μ has time t'.

Definition 2. *$(x, t) \rightsquigarrow (x', t')$ if and only if*

- *$x = x'$ and $t \leq t'$, or*
- *there exist $k > 1$, distinct x_i for $i = 1 \ldots k$, and (not necessarily distinct) t_i for $i = 1 \ldots k$, such that $x = x_1$, $x' = x_k$, $t \leq t_1$, and $t_k \leq t'$, and for all $i = 1 \ldots k - 1$:*
  - *$x_i \in I \cup E$, $x_{i+1} \in P$, $dst(x_i) = x_{i+1}$, and $t_i = t_{i+1}$, or*
  - *$x_i \in P$, $x_{i+1} \in E \cup O$, $src(x_{i+1}) = x_i$, and there exist $t'_i \geq t_i$ and $t''_i \leq t_{i+1}$ such that $(x_i, t'_i) \rightsquigarrow^1 (x_{i+1}, t''_i)$.*

In the first case, we say that the proof of $(x, t) \rightsquigarrow (x', t')$ has length 1; in the second, that it has length k. (These lengths are helpful in inductive arguments. Different proofs of $(x, t) \rightsquigarrow (x', t')$ may in general have different lengths.)

This definition captures the semantics of an arbitrary node p, via the functions g_1 and g. The function g is applied to a local history to generate a state s, then g_1 is applied at s. Thus, the definition restricts attention to states s that can arise in some execution with p. However, we do not attempt to guarantee that this execution is one of those that can occur in the context of the other nodes in the graph of interest, in order to avoid a circularity: this latter set of executions is itself defined in terms of the relation $\rightsquigarrow$ (in Section 4.2).

An implementation, such as Naiad's, may soundly use simple, conservative approximations to the relation $\rightsquigarrow$ as we define it here. In Naiad, for most nodes p, it is assumed that $(p, t) \rightsquigarrow^1 (e, t)$ for all t and each outgoing edge e; certain nodes (loop ingress, feedback, and egress) receive special treatment.

The definition implies that $\rightsquigarrow$ is reflexive. The following proposition asserts a few of the additional properties of $\rightsquigarrow$ that we have found useful.

Proposition 1. *1. If $(p, t_1) \rightsquigarrow (e, t_2)$ then there are $e' \in E \cup O$ and $t' \in T$ such that $src(e') = p$, $(p, t_1) \rightsquigarrow (e', t')$ and $(e', t') \rightsquigarrow (e, t_2)$, with the proof of $(e', t') \rightsquigarrow (e, t_2)$ strictly shorter than that of $(p, t_1) \rightsquigarrow (e, t_2)$.*

2. *If* $(x, t_1) \rightsquigarrow (x, t_2)$ *then* $t_1 \leq t_2$.
3. *If* $(x_1, t_1) \rightsquigarrow (x_2, t_2)$, $t'_1 \leq t_1$, *and* $t_2 \leq t'_2$, *then* $(x_1, t'_1) \rightsquigarrow (x_2, t'_2)$.

The definition is designed to be convenient in proofs and to reflect important aspects of implementations (and of Naiad's specifically). In particular, the distinctness requirement ("there exist $k > 1$, distinct x_i for $i = 1 \dots k$") means that proofs and implementations do not need to chase around cycles.

On the other hand, because of the distinctness requirement, the definition does not immediately yield that $\rightsquigarrow$ is transitive, as one might expect, and as one might often want in proofs. More broadly, the definition of $\rightsquigarrow$ may not correspond to the intuitive understanding of could-result-in without some additional assumptions, which we address next.

3.2 On Sending Notification Requests and Messages into the Past

In timely dataflow, and in Naiad in particular, it is generally expected that events do not give rise to other events at earlier times. When those other events are notification requests, the required condition is easy to state. When they are messages, it is not, because we do not wish to compare times across time domains. In this section we formulate and study these two conditions.

The first considers the generation of notification requests, which the definition of $\rightsquigarrow$ ignores. We formulate it via an additional relation $\rightsquigarrow_{\mathrm{N}}$, a local variant of the could-result-in relation that focuses on the generation of notification requests. (This relation is not intended to be reflexive or transitive.)

Definition 3. $(p, t) \rightsquigarrow_{\mathrm{N}} (p, t')$ *if and only if there exist a history* h *for* p *and a state* s *such that*

$$g(p)(h) = (s, N_1, \dots)$$

and an event x *such that either* $x = t''$ *for some* t'' *such that* $t \leq t''$, *or* $x = (e, m)$ *for some* e *and* m *such that* $t \leq \mathit{time}(m)$, *and*

$$g_1(p)(s, x) = (\dots, N, \dots)$$

where some element of $N - N_1$ *is* $\leq t'$.

Using this relation, we can express that an event at time t can trigger notification requests only at greater times t':

Condition 1 *If* $(p, t) \rightsquigarrow_{\mathrm{N}} (p, t')$ *then* $t \leq t'$.

The question of the transitivity of $\rightsquigarrow$ is closely related to the expectation that nodes should not be allowed to send messages into the past. Indeed, a sufficient condition for transitivity is that, for all pointstamps (x, t) and (x', t'), if $(x, t) \rightsquigarrow (x', t')$ then $t \leq t'$ (as implied by Theorem 1, below). However, the converse does not hold, for trivial reasons. For example, in a graph with a single node, the relation $\rightsquigarrow$ will always be transitive but we may not have that $(x, t) \rightsquigarrow (x', t')$ implies $t \leq t'$. Still, we can compare times at a node and at its incoming edges, and fortunately such comparisons suffice, as the following theorem demonstrates.

Condition 2 *For all $p \in P$, $e \in E$ with $dst(e) = p$, and $t, t' \in T$, if $(p, t) \rightsquigarrow (e, t')$ then $t \leq t'$.*

Theorem 1. *The relation $\rightsquigarrow$ is transitive if and only if Condition 2 holds.*

Conditions 1 and 2 both depend on the semantics of individual nodes; Condition 2 also depends on the topology of the graph. Although we assume them in some of our results, we do not discuss how they can be enforced. In practice, Naiad simply assumes analogous properties, but type systems and other static analyses may well help in checking them.

The following proposition offers another way of thinking about transitivity by comparing times at different nodes and edges, via an embedding of these times into an additional partial order $(T', \preceq)$. One may view $(T', \preceq)$ as a set of times normalized into a coherent universal time—the "GMT" of timely dataflow. (This proposition is fairly straightforward, and we do not need it below.)

Proposition 2. *The relation $\rightsquigarrow$ is transitive if and only if there exist a partial order $(T', \preceq)$ and a mapping $\mathcal{E}$ from the set of pointstamps $((I \cup E \cup O) \cup P) \times T$ to T' such that, for all (x, t) and (x', t'), $(x, t) \rightsquigarrow (x', t')$ if and only if $\mathcal{E}(x, t) \preceq \mathcal{E}(x', t')$.*

3.3 Closure

We say that a set S of pointstamps is *upward closed* if and only if, for all pointstamps (x, t) and (x', t'), $(x, t) \in S$ and $(x, t) \rightsquigarrow (x', t')$ imply $(x', t') \in S$. For any set S of pointstamps, $Close_{\uparrow}(S)$ is the least upward closed set that contains S. Assuming that $\rightsquigarrow$ is transitive, the following proposition provides a simpler formulation for $Close_{\uparrow}$.

Proposition 3. *Assume that Condition 2 holds. Then $Close_{\uparrow}(S) = \{(x', t') \mid \exists (x, t) \in S. (x, t) \rightsquigarrow (x', t')\}$.*

3.4 Recursion

Naiad focuses on iterative computation, and the could-result-in relation for the nodes that support iteration (loop ingress, feedback, and egress) has been discussed informally [12]. We could revisit iteration using our definitions. However, the definitions are much more general. We demonstrate the value of this generality by outlining how they apply to recursive dataflow computation (e.g., [5]).

Let us consider a dataflow graph that includes a distinguished input node `in` with no incoming edges, a distinguished output node `out` with no outgoing edges, some ordinary nodes for operations on data, and other nodes that represent recursive calls to the entire computation. For simplicity, we let $I = O = \emptyset$, and do not consider multiple mutually recursive graphs and other variants. We assume that every node is reachable from `in`, `out` is reachable from every node, and there is a path from `in` to `out` that does not go through any call nodes. In order to make the recursion explicit, we modify the dataflow graph by splitting

each call node `c` into a call part `call-c` and a return part `ret-c`, where the former is the source of a back edge to `in` and the latter is the destination of a back edge from `out`.

A stack $\int$ is a finite sequence of call nodes. We let $\dot{\rightsquigarrow}$ be the least reflexive, transitive relation on pairs $(p, \int)$ such that

1. $(\texttt{call-c}, \int) \dot{\rightsquigarrow} (\texttt{in}, \int \cdot \texttt{c})$;
2. symmetrically, $(\texttt{out}, \int \cdot \texttt{c}) \dot{\rightsquigarrow} (\texttt{ret-c}, \int)$; and
3. if p is not `call-c` and p' is not `ret-c` for any `c`, and there is an edge from p to p', then $(p, \int) \dot{\rightsquigarrow} (p', \int)$.

We will have that $\dot{\rightsquigarrow}$ is a conservative approximation of $\rightsquigarrow$.

At each node p, we define a pre-order on stacks: $\int \preceq_p \int'$ if and only if $(p, \int) \dot{\rightsquigarrow} (p, \int')$. We write $(T_p, \leq_p)$ for the partial order induced by $\preceq_p$ (so, T_p identifies $\int$ and $\int'$ when both $\int \preceq_p \int'$ and $\int' \preceq_p \int$). The partial order is thus different at each node. The partial order of virtual times $(T, \leq)$ is the disjoint ("tagged") union of the partial orders $(T_p, \leq_p)$ for all the nodes. We write $[\int]_p$ for the element of T obtained by tagging the equivalence class of $\int$ at p.

We assume that each node p uses the appropriate tags for its outgoing messages and notification requests, and ignores inputs and notifications not tagged with p, and also that the behavior of p, as reflected in the relation $\rightsquigarrow^1$, conforms to what the relation $\dot{\rightsquigarrow}$ expresses:

If $(p, [\int]_q) \rightsquigarrow^1 (d, [\int']_{p'})$ then $q = p$, $p' = dst(d)$, and $(p, \int) \dot{\rightsquigarrow} (p', \int')$.

We obtain:

Proposition 4. *If* $(p, [\int]_p) \rightsquigarrow (p', [\int']_{p'})$ *then* $(p, \int) \dot{\rightsquigarrow} (p', \int')$.

Proposition 5. *If* $(p, [\int]_q) \rightsquigarrow (p', [\int']_{q'})$ *and* $p \neq p'$, *then* $q = p$ *and* $q' = p'$.

Applying Theorem 1, we also obtain:

Proposition 6. *The relation* $\rightsquigarrow$ *is transitive.*

Furthermore, the relation $\dot{\rightsquigarrow}$ can be decided quite simply by finding the first call in which two stacks differ and performing an easy check based on that difference. This check relies on an alternative modified graph, in which we split each call node `c` into a call part `call-c` and a return part `ret-c`, but add a direct forward edge from the former to the latter (rather than back edges). Suppose (without loss of generality) that $\int$ is of the form $\int_1 \cdot \int_2$ and $\int'$ is of the form $\int_1 \cdot \int_2'$, where $\int_2$ and $\int_2'$ start with `c` and `c'` respectively if they are not empty. We assume that `c` and `c'` are distinct if $\int_2$ and $\int_2'$ are both non-empty (so, $\int_1$ is maximal). Let l be `ret-c` if $\int_2$ is non-empty, and be p if it is empty; let l' be `call-c'` if $\int_2'$ is non-empty, and be p' if it is empty. Then we can prove that $(p, \int) \dot{\rightsquigarrow} (p', \int')$ if and only if there is a path from l to l' in the alternative modified graph.

Special cases (in particular, special graph topologies) may allow further simplifications which could be helpful in implementations.

4 Semantics

We describe the semantics of timely dataflow graphs in a state-based framework [3, 11]. In this section, we first review this framework, then specify the semantics. Finally, we discuss matters of compositionality.

4.1 The Framework (Review)

The sequence $\langle\!\langle s_0, s_1, s_2, \ldots \rangle\!\rangle$ is said to be *stutter-free* if, for each i, either $s_i \neq s_{i+1}$ or the sequence is infinite and $s_i = s_j$ for all $j \geq i$. We let $\natural\sigma$ be the stutter-free sequence obtained from σ by replacing every maximal finite subsequence $s_i, s_{i+1}, \ldots, s_j$ of identical elements with the single element s_i. A set of sequences S is *closed under stuttering* when $\sigma \in S$ if and only if $\natural\sigma \in S$.

A *state space* Σ is a subset of $\Sigma_E \times \Sigma_I$ for some sets Σ_E of externally visible states and Σ_I of internal states. If Σ is a state space, then a *Σ-behavior* is an element of Σ^ω. A Σ_E-behavior is called an *externally visible behavior*. A *Σ-property* P is a set of Σ-behaviors that is closed under stuttering. When Σ is clear from context or is irrelevant, we may leave it implicit. We sometimes apply the adjective "complete", as in "complete behavior", in order to distinguish behaviors and properties from externally visible behaviors and properties.

A *state machine* is a triple (Σ, F, N) where Σ is a state space; F, the set of *initial* states, is a subset of Σ; and N, the *next-state relation*, is a subset of $\Sigma \times \Sigma$. The *complete property generated by* a state machine (Σ, F, N) consists of all infinite sequences $\langle\!\langle s_0, s_1, \ldots \rangle\!\rangle$ such that $s_0 \in F$ and, for all $i \geq 0$, either $\langle s_i, s_{i+1} \rangle \in N$ or $s_i = s_{i+1}$. The *externally visible property generated by* a state machine is the externally visible property obtained from its complete property by projection onto Σ_E and closure under stuttering. For brevity, we do not consider fairness conditions or other liveness properties that can be added to state machines; their treatment is largely orthogonal to our present goals.

Although we are not fully formal in the use of TLA [11], we generally follow its approach to writing specifications. Specifically, we express state machines by formulas of the form:

$$\exists y_1, \ldots, y_n.\ F \wedge \Box[N]_{v_1,\ldots,v_k}$$

where:

- state functions that we write as variables represent the state;
- we distinguish external variables and internal variables, and the internal variables (in this case, $y_1, \ldots, y_n$) are existentially quantified;
- F is a formula that may refer to the variables;
- $\Box$ is the temporal-logic operator "always";
- N is a formula that may refer to the variables and also to primed versions of the variables (thus denoting the values of those variables in the next state);
- $[N]_{v_1,\ldots,v_k}$ abbreviates $N \vee ((v_1' = v_1) \wedge \ldots \wedge (v_k' = v_k))$.

4.2 Semantics Specification

In our semantics, the externally visible states map each $e \in I \cup O$ to a value $Q(e)$ in M^*. In other words, we observe only the state of input and output edges. The internally visible states map each $e \in E$ to a value $Q(e)$ in M^*, and each $p \in P$ to a local state $\mathit{LocState}(p) \in \Sigma_{\mathrm{Loc}}$ and to a set of pending notification requests $\mathit{NotRequests}(p) \in \mathcal{P}(T)$.

An auxiliary state function *Clock* (whose name comes from Naiad, and is unrelated to "clocks" elsewhere) tracks pointstamps for which work may remain:

$$\mathit{Clock} \triangleq \mathit{Close}_{\uparrow}\left(\begin{array}{c}\{(e, \mathit{time}(m)) \mid e \in I \cup E \cup O, m \in Q(e)\} \\ \cup \\ \{(p,t) \mid p \in P, t \in \mathit{NotRequests}(p)\}\end{array}\right)$$

We define an initial condition, the actions that constitute a next-state relation, and finally the specification.

Initial condition:

$$\mathit{InitProp} \triangleq \left(\begin{array}{l}\forall e \in E \cup O.Q(e) = \emptyset \wedge \forall i \in I.Q(i) \in M^* \\ \wedge \\ \forall p \in P.(\mathit{LocState}(p), \mathit{NotRequests}(p)) \in \mathit{Initial}(p)\end{array}\right)$$

Actions:

1. Receiving a message:

$$\mathit{Mess} \triangleq \exists p \in P.\mathit{Mess1}(p)$$

$$\mathit{Mess1}(p) \triangleq \left(\begin{array}{l}\exists m \in M.\exists e \in I \cup E \text{ such that } p = \mathit{dst}(e). \\ Q(e) = m \cdot Q'(e) \wedge \mathit{Mess2}(p, e, m)\end{array}\right)$$

$$\mathit{Mess2}(p, e, m) \triangleq \left(\begin{array}{l}\mathit{let} \\ \{e_1, \ldots, e_k\} = \{d \in E \cup O \mid \mathit{src}(d) = p\}, \\ s = \mathit{LocState}(p), \\ (s', N, \langle e_1 \mapsto \mu_1, \ldots, e_k \mapsto \mu_k \rangle) = g_1(p)(s, (e, m)) \\ \mathit{in} \\ \mathit{LocState}'(p) = s' \\ \wedge \\ \mathit{NotRequests}'(p) = \mathit{NotRequests}(p) \cup N \\ \wedge \\ Q'(e_1) = Q(e_1) \cdot \mu_1 \ldots Q'(e_k) = Q(e_k) \cdot \mu_k \\ \wedge \\ \forall q \in P \neq p.\mathit{LocState}'(q) = \mathit{LocState}(q) \\ \wedge \\ \forall q \in P \neq p.\mathit{NotRequests}'(q) = \mathit{NotRequests}(q) \\ \wedge \\ \forall d \in I \cup E \cup O - \{e, e_1, \ldots, e_k\}.Q'(d) = Q(d)\end{array}\right)$$

These formulas describe how a node p dequeues a message m and reacts to it, producing messages and notification requests.

2. Receiving a notification:

$$Not \triangleq \exists p \in P.Not1(p)$$

$$Not1(p) \triangleq \left(\begin{array}{l} \exists t \in NotRequests(p). \\ \forall e \in I \cup E \text{ such that } dst(e) = p.(e,t) \notin Clock \\ \wedge \\ Not2(p,t) \end{array} \right)$$

$$Not2(p,t) \triangleq \left(\begin{array}{l} let \\ \{e_1, \ldots, e_k\} = \{d \in E \cup O \mid src(d) = p\}, \\ s = LocState(p), \\ (s', N, \langle e_1 \mapsto \mu_1, \ldots, e_k \mapsto \mu_k \rangle) = g_1(p)(s,t) \\ in \\ LocState'(p) = s' \\ \wedge \\ NotRequests'(p) = NotRequests(p) - \{t\} \cup N \\ \wedge \\ Q'(e_1) = Q(e_1) \cdot \mu_1 \ldots Q'(e_k) = Q(e_k) \cdot \mu_k \\ \wedge \\ \forall q \in P \neq p.LocState'(q) = LocState(q) \\ \wedge \\ \forall q \in P \neq p.NotRequests'(q) = NotRequests(q) \\ \wedge \\ \forall d \in I \cup E \cup O - \{e_1, \ldots, e_k\}.Q'(d) = Q(d) \end{array} \right)$$

These formulas describe how a node p consumes a notification t for which it has an outstanding notification request, and how it reacts to the notification, producing messages and notification requests.

3. External input and output changes:

$$Inp \triangleq \left(\begin{array}{l} \forall i \in I.Q(i) \text{ is a subsequence of } Q'(i) \\ \wedge \\ \forall p \in P.LocState'(p) = LocState(p) \\ \wedge \\ \forall p \in P.NotRequests'(p) = NotRequests(p) \\ \wedge \\ \forall d \in E \cup O.Q'(d) = Q(d) \end{array} \right)$$

$$Outp \triangleq \left(\begin{array}{l} \forall o \in O.Q'(o) \text{ is a subsequence of } Q(o) \\ \wedge \\ \forall p \in P.LocState'(p) = LocState(p) \\ \wedge \\ \forall p \in P.NotRequests'(p) = NotRequests(p) \\ \wedge \\ \forall d \in I \cup E.Q'(d) = Q(d) \end{array} \right)$$

External input changes allow the contents of input edges to be extended rather arbitrarily. We do not assume that such extensions are harmonious with notifications and the use of *Clock*; from this perspective, it would be reasonable and straightforward to add the constraint $Clock' \subseteq Clock$ to *Inp*. Similarly, external output changes allow the contents of output edges to be removed, not necessarily in order. We ask that $Q(i)$ be a subsequence of $Q'(i)$ and that $Q'(o)$ be a subsequence of $Q(o)$, so that it is easy to attribute state transitions. While variants on these two actions are viable, allowing some degree of external change to input and output edges seems attractive for composability (see Section 4.3).

The high-level specification:

$$\begin{aligned} ISpec &\triangleq InitProp \wedge \Box[Mess \vee Not \vee Inp \vee Outp]_{LocState, NotRequests, Q} \\ Spec &\triangleq \exists LocState, NotRequests, Q \upharpoonright E.ISpec \end{aligned}$$

ISpec describes a complete property and *Spec* an externally visible property.

This specification is the most basic of several that we have studied. For instance, another one allows certain message reorderings, replacing *Mess1*(p) with

$$\begin{aligned} &\exists m \in M. \exists e \in I \cup E \text{ such that } p = dst(e). \exists u, v \in M^*. \\ &Q(e) = u \cdot m \cdot v \wedge Q'(e) = u \cdot v \wedge \forall n \in u.time(n) \not\leq time(m) \wedge Mess2(p, e, m) \end{aligned}$$

Given a queue of messages $Q(e)$, p is allowed to process any message m such that there is no message n ahead of m with $time(n) \leq time(m)$. Mathematically, we may think of $Q(e)$ as a partially ordered multiset (pomset) [13]; with that view, m is a minimal element of $Q(e)$. This relaxation is useful, for example, for enabling optimizations in which several messages for the same time are processed together, even if they are not all at the head of a queue.

4.3 Composing Graphs

We briefly discuss how to compose graphs, without however fully developing the corresponding definitions and theory (in part, simply, because we have not needed them in our applications of the semantics to date).

We can regard the specifications of this paper as being parameterized by a *Clock* variable, rather than as being specifically for *Clock* as defined in Section 4.2. Once we regard *Clock* as a parameter, the specifications that correspond to multiple dataflow graphs can be composed meaningfully, and along standard lines [9]. Suppose that we are given graphs G_1 and G_2, with nodes P_1 and P_2, input edges I_1 and I_2, internal edges E_1 and E_2, and output edges O_1 and O_2, and specifications $Spec_1$ and $Spec_2$. We assume that P_1 and P_2, I_1 and I_2, E_1 and E_2, and O_1 and O_2 are pairwise disjoint. We also assume that I_1, I_2, O_1, and O_2 are disjoint from E_1 and E_2. We write $X_{12} = I_2 \cap O_1$ and $X_{21} = I_1 \cap O_2$. The edges in X_{12} and X_{21} will connect the two graphs. We define a specification for the

composite system with nodes $P = P_1 \cup P_2$, input edges $I = I_1 \cup I_2 - X_{12} - X_{21}$, internal edges $E = E_1 \cup E_2 \cup X_{12} \cup X_{21}$, and output edges $O = O_1 \cup O_2 - X_{12} - X_{21}$, by

$$\mathit{Spec}_{12} = \exists Q \restriction (X_{12} \cup X_{21}).\mathit{Spec}_1 \wedge \mathit{Spec}_2 \wedge \Box[\neg(\mathit{Acts}_1 \wedge \mathit{Acts}_2)]$$

where, for $j = 1, 2$,

$$\mathit{Acts}_j = \left[\begin{array}{l} \exists i \in I_j.Q'(i) \text{ is a proper subsequence of } Q(i) \\ \vee \\ \exists o \in O_j.Q(o) \text{ is a proper subsequence of } Q'(o) \end{array}\right]$$

The formula $\Box[\neg(\mathit{Acts}_1 \wedge \mathit{Acts}_2)]$ ensures that the actions of the two subsystems that are visible on their input and output edges are not simultaneous. It does not say anything about internal edges, nor does it address notification requests.

It remains to study how Spec_{12} relates to the non-compositional specification of the same system. Going further, the definition of a global *Clock* might be obtained compositionally from multiple, more local could-result-in relations. Finally, one might address questions of full abstraction. Although we rely on a state-based formalism, results such as Jonsson's [9] (which are cast in terms of I/O automata) should translate. However, a fully abstract treatment of timely dataflow would have interesting specificities, such as the handling of completion notifications and the possible restrictions on contexts (in particular contexts constrained not to send messages into the past).

5 An Application

In order to leverage the definitions and to test them, we state and prove a basic but important property for timely dataflow. Specifically, we argue that, once a pointstamp (e, t) is not in *Clock*, messages on e will never have times $\leq t$. For this property to hold, however, we require a hypothesis on inputs; we simply assume that, for all input edges i, $Q(i)$ never grows, though it may contain some messages initially. (Alternatively, we could add the constraint $\mathit{Clock}' \subseteq \mathit{Clock}$ to *Inp*, as suggested in Section 4.2.)

First, we establish some auxiliary propositions:

Proposition 7. *ISpec implies that always, for all $p \in P$, there exists a local history $H(p)$ for p such that $\mathit{LocState}(p) = \Pi_{\mathrm{Loc}} g(H(p))$ and $\mathit{NotRequests}(p) = \Pi_{\mathrm{NR}} g(H(p))$.*

Proposition 8. *Assume that Conditions 1 and 2 hold. Then ISpec implies*

$$\Box[(\forall i \in I.Q'(i) \textit{ is a subsequence of } Q(i)) \Rightarrow (\mathit{Clock}' \subseteq \mathit{Clock})]$$

Proposition 9.

$$\begin{array}{l}\Box[(\forall i \in I.Q'(i) \textit{ is a subsequence of } Q(i)) \Rightarrow (\mathit{Clock}' \subseteq \mathit{Clock})] \\ \quad \wedge \\ \Box[\forall i \in I.Q'(i) \textit{ is a subsequence of } Q(i)] \\ \quad \Rightarrow \\ \Box \forall e \in I \cup E \cup O, t \in T. \left[\begin{array}{l}(e,t) \notin \mathit{Clock} \\ \quad \Rightarrow \\ \Box(e,t) \notin \mathit{Clock}\end{array}\right]\end{array}$$

We obtain:

Theorem 2. *Assume that Conditions 1 and 2 hold. Then ISpec and*

$$\Box[\forall i \in I.Q'(i) \textit{ is a subsequence of } Q(i)]$$

imply

$$\Box \forall e \in I \cup E \cup O, t \in T, m \in M. \left[\begin{array}{l}(e,t) \notin \mathit{Clock} \\ \quad \Rightarrow \\ \Box(m \in Q(e) \Rightarrow \mathit{time}(m) \not\leq t)\end{array}\right]$$

Previous work [4] studies a distributed algorithm for tracking the progress of a computation, and arrives at a somewhat analogous result. This previous work assumes a notion of virtual time but defines neither a dataflow model nor a corresponding could-result-in relation (so, in particular, it does not treat analogues of Conditions 1 and 2). In the distributed algorithm, information at each processor serves for constructing a conservative approximation of the pending work in a system. Naiad relies on such an approximation for implementing its clock, which the state function *Clock* represents in our model.

6 Conclusion

This paper aims to develop a rigorous foundation for timely dataflow, a model for data-parallel computing. Some of the ingredients in timely dataflow, as defined in this paper, have a well-understood place in the literature on semantics and programming languages. For instance, many programming languages support messages and message streams. On the other hand, despite similarities to extant concepts, other ingredients are more original, so giving them self-contained semantics can be both interesting and valuable for applications. In particular, virtual times and completion notifications may be reminiscent of the notion of priorities [15, 8], but a straightforward reduction seems impossible. More broadly, there should be worthwhile opportunities for further foundational and formal contributions to research on data-parallel software, currently a lively area of experimental work in which several computational abstractions and models are being revisited, adapted, or invented.

Acknowledgments

We are grateful to our coauthors on work on Naiad for discussions that led to this paper. In addition, conversations with Nikhil Swamy and Dimitrios Vytiniotis motivated our study of recursion.

References

1. Abadi, M., Isard, M.: On the flow of data, information, and time. In: Focardi, R., Myers, A. (eds.) Proceedings of the 4th Conference on Principles of Security and Trust. Springer (2015), to appear.
2. Abadi, M., Isard, M.: Timely rollback: Specification and verification. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) Proceedings of the 7th NASA Formal Methods Symposium. Springer (2015), to appear.
3. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science 82(2), 253–284 (1991)
4. Abadi, M., McSherry, F., Murray, D.G., Rodeheffer, T.L.: Formal analysis of a distributed algorithm for tracking progress. In: Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013. pp. 5–19 (2013)
5. Blelloch, G.E.: Programming parallel algorithms. Communications of the ACM 39(3), 85–97 (Mar 1996)
6. Hildebrandt, T.T., Panangaden, P., Winskel, G.: A relational model of non-deterministic dataflow. Mathematical Structures in Computer Science 14(5), 613–649 (2004)
7. Jefferson, D.R.: Virtual time. ACM Transactions on Programming Languages and Systems 7(3), 404–425 (Jul 1985)
8. John, M., Lhoussaine, C., Niehren, J., Uhrmacher, A.M.: The attributed pi-calculus with priorities. Transactions on Computational Systems Biology 12, 13–76 (2010)
9. Jonsson, B.: A fully abstract trace model for dataflow and asynchronous networks. Distributed Computing 7(4), 197–212 (1994)
10. Kahn, G.: The semantics of a simple language for parallel programming. In: IFIP Congress. pp. 471–475 (1974)
11. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
12. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: a timely dataflow system. In: ACM SIGOPS 24th Symposium on Operating Systems Principles. pp. 439–455 (2013)
13. Pratt, V.: Modeling concurrency with partial orders. International Journal of Parallel Programming 15(1), 33–71 (1986)
14. Selinger, P.: First-order axioms for asynchrony. In: Mazurkiewicz, A.W., Winkowski, J. (eds.) CONCUR '97: Concurrency Theory, 8th International Conference. vol. 1243, pp. 376–390. Springer (1997)
15. Versari, C., Busi, N., Gorrieri, R.: An expressiveness study of priority in process calculi. Mathematical Structures in Computer Science 19(6), 1161–1189 (2009)