Logical Time for Reactive Software

Marten Lohstroh
marten@berkeley.edu
University of California, Berkeley
USA

Edward A. Lee
eal@berkeley.edu
University of California, Berkeley
USA

Stephen A. Edwards
sedwards@cs.columbia.edu
Columbia University, New York, NY
USA

David Broman
dbro@kth.se
KTH Royal Institute of Technology, Stockholm
Sweden

ABSTRACT

Timing is an essential feature of reactive software. It is not just a performance metric, but rather forms a core part of the semantics of programs. This paper argues for a notion of *logical time* that serves as an engineering model to complement a notion of *physical time*, which models the physical passage of time. Programming models that embrace logical time can provide deterministic concurrency, better analyzability, and practical realizations of timing-sensitive applications. We give definitions for physical and logical time and review some languages and formalisms that embrace logical time.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

timing, reactive systems, programming model, software

ACM Reference Format:
Marten Lohstroh, Edward A. Lee, Stephen A. Edwards, and David Broman. 2023. Logical Time for Reactive Software. In *Cyber-Physical Systems and Internet of Things Week 2023 (CPS-IoT Week Workshops '23), May 09–12, 2023, San Antonio, TX, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3576914.3587494

1 INTRODUCTION

Timing of software execution is usually considered a performance property rather than a correctness property. But in software for cyber-physical systems, timing is often a critical feature of the execution of the software. Today, no widely used programming language specifies timing. Instead, timing is an emergent consequence of a particular implementation and is sensitive to every detail of the hardware on which the software runs and to what other software may be sharing the same hardware. Even a small change in the hardware or software context can lead to drastically different timing behavior, making testing, maintenance, and upgrades difficult.

Just as a programmer delegates to the compiler and the microprocessor correct execution of the program logic, we argue that there should be programmatic ways to similarly delegate delivery of timing requirements. In recent years, inroads have been made towards realizing this vision. But until the day that abstractions for time are available in mature and well-supported programming languages, compilers, and computer and network architectures, significant challenges remain.

In this paper, we discuss logical time as a means for establishing a sound engineering practice around the development of time-centric reactive software. The main contributions are (i) precise definitions of physical and logical time, and (ii) a short review of some languages that embrace logical time.

2 HOW TO MODEL TIME

Software should employ an *engineering model* of time that can be implemented in practice and reasoned about by humans instead of a *scientific model* that models physical reality [20]. The classical Newtonian model of time, which assumes there is a global state of the system that is known instantaneously everywhere, is a good approximation for relatively slow, relatively local, continuous dynamics, but today's electronic systems may span the globe, operate with sub-nanosecond timing, and consist of discrete, discontinuous state transitions. These systems are not Newtonian because the order in which physically separated events occur is neither practically knowable nor theoretically well-defined [30, 34]. Distributed systems have no well-defined "current state."

2.1 Logical Time vs. Physical Time

We want an unambiguous order of events because application behavior often depends on such ordering (think of bank account transactions), but ordering defies simple observation [21]. If two geographically separated components each perform an event nearly simultaneously, the "true" order of events may depend on your frame of reference. As a consequence, reality precludes a knowable current state of a distributed system. There is more than one truth.

Assigning logical *timestamps* can *impose* a well-defined event ordering. If two separated components assign to their respective events two timestamps t_1 and t_2 drawn from a totally ordered set $\mathbb{T}$, we can have a clear, unambiguous *semantic* model of the progression of the system based on the order of these timestamps. This is not a scientific model because we do not demand the ordering of timestamps necessarily match any physical truth, but it is an engineering model that we can implement faithfully. In particular,

CPS-IoT Week Workshops '23, May 09–12, 2023, San Antonio, TX, USA

ACM ISBN 979-8-4007-0049-1/23/05.
https://doi.org/10.1145/3576914.3587494

provided the system components all agree on events' timestamps, the system will agree on events' global ordering.

Distributed databases systems often take this approach, assigning timestamps based on *local clocks*. These clocks are imperfect by *any* physical model of time, but are often adequate. An operating system clock synchronized by NTP [29] often suffices; more demanding applications use GPS, PTP [10], or atomic clocks. But these choices only affect utility, not correctness.

Using timestamps proposes a *logical* model of time that is consistent regardless of how the timestamps are assigned, provided they are drawn from a totally ordered set. They need not even be related at all to any wall clock measuring physical time. For an implementation to be correct (faithful to its semantics), it only needs to ensure all observers see the events in timestamp order. The key advantage here is a correctness criterion that completely sidesteps the question of how to interpret physical reality, providing us with a means to unambiguously specify timed behavior. But for such software to be *useful*, we usually want to be able to relate timestamps to the passage of time as perceived by physical observers.

Assume that physical time at a single point in space behaves like a smoothly advancing real number. Assume that such time is measured by a clock $c \in C$ located at that point in space, where C is a set of clocks. Every clock will be imperfect, so we do not assume consistency between distinct clocks.

Definition 2.1 (Physical time). Let $\mathbb{T}$ be the set of **physical time values** that a clock $c \in C$ may return. The set $\mathbb{T}$ is totally ordered. When a clock c attempts to measure a time instant $\tau \in \mathbb{R}$, it returns a $\mathcal{P}_c(\tau) \in \mathbb{T}$ given by **physical time** function

$$\mathcal{P}_c \colon \mathbb{R} \to \mathbb{T}.$$

The various $\mathcal{P}_c$ and the underlying τ values they measure are unknown to the system (and unknowable).

The quality of the physical clocks in a system dictates the properties of the various $\mathcal{P}_c$ functions, which always fall short of ideal. We may desire successive interrogations of the same clock yield strictly larger measurements, i.e., $\tau_1 < \tau_2$ implies $\mathcal{P}_c(\tau_1) < \mathcal{P}_c(\tau_2)$, but this is practically difficult. First, real clocks report quantized time values, so at best, $\mathcal{P}_c(\tau_1) \leq \mathcal{P}_c(\tau_2)$. Even worse, certain clocks occasionally run backward, e.g., an operating system clock being adjusted by NTP.

It is often useful to endow the set $\mathbb{T}$ with a metric, a distance function $d \colon \mathbb{T} \times \mathbb{T} \to \mathbb{R}$ with the usual properties of a metric. This enables quantifying the passage of time rather than just counting.

A measurement m of a system's environment (e.g., a sensor reading) may be marked with a reading $T_m = \mathcal{P}_c(\tau) \in \mathbb{T}$ from a local physical clock c taken when the measurement is taken (at time τ). Although τ cannot be directly known, this strategy ensures that readings from the same sensor will be treated consistently throughout a system regardless of how its other clocks may behave.

Systems may employ an additional level of abstraction between physical time values and the logical time values in their semantics:

Definition 2.2 (Tags and Logical time). A tag g_e for an event e is a member of a totally ordered set $\mathbb{G}$ endowed with a monotonic function $\mathcal{T} \colon \mathbb{G} \to \mathbb{T}$. The tag denotes a **logical time**; any two events that have equal tags are **logically simultaneous**. The **timestamp** $\mathcal{T}(g_e)$ of the event e can be compared directly against a physical time obtained from a clock because it is a member of the same totally ordered set $\mathbb{T}$.

In the simplest case, $\mathbb{G} = \mathbb{T}$ and $\mathcal{T}$ is the identity function, but in general, the ordering of events may need to be controlled in a more or less fine-grained way than is possible using the set $\mathbb{T}$ of time readings that a physical clock may yield. This is why the tag set $\mathbb{G}$ is not required to match time values that may be yielded by a physical clock. To tag a sensor reading taken at (unknown) time τ, we select a tag $g_m \in \mathbb{G}$ such that $\mathcal{T}(g) = \mathcal{P}_c(\tau)$, where c is a local physical clock.

Our tags are not as general as those of Lee and Sangiovanni-Vincentelli [23]; the set $\mathbb{G}$ is required to be totally ordered so that each tag can interpreted as a logical time value, and the function $\mathcal{T}$ gives a way to relate that logical time value to physical time.

2.2 Representations of Time

Choosing the set of physical time values $\mathbb{T}$ and the set of tags $\mathbb{G}$ must be done judiciously to achieve reasonable fidelity between the physical world and the logical world of the system. Here, we discuss considerations and choices.

Time as a real number may be questioned since some physical theories posit a discrete version of time that is granular on the order of Planck time (5.39×10^{-44} s), but this is not relevant for engineering and we know of no system with a clock this precise. This is why we use a real number τ to represent the (unknowable) physical time. This makes it tempting to represent time in software systems with floating-point numbers, commonly used to approximate real numbers. But this turns out to be a poor choice, as explained by Broman, et al. [6]. Integer representations prove to be better.

The synchronous languages (see Section 3 below) use $\mathbb{G} = \mathbb{N}$, the natural numbers, to count the number of "instants" the program has encountered, and the mapping to *physical* clock readings is an arbitrary monotonic function that is irrelevant to the program correctness.

Sometimes, a more fine-grained mechanism is useful. For example, situations can arise where two events with the same timestamp $t \in \mathbb{T}$ are causally related and should not be treated as being logically simultaneous. In such situations, it has proven useful to use superdense time [7, 27, 28] or non-standard time [2].

Superdense Time. A superdense time model may use $\mathbb{G} = \mathbb{T} \times \mathbb{N}$ where $\mathbb{N}$ is the natural numbers, and for any $g = (t, n) \in \mathbb{G}$, $\mathcal{T}(g) = t$. That is, an event has a tag (t, n), where t is the timestamp and n is a superdense time index. In such a model, the total order relation used is the dictionary order, where $(t_1, n_1) < (t_2, n_2)$ if $t_1 < t_2$ or $t_1 = t_2$ and $n_1 < n_2$. This model allows for causally related events to have distinct tags without any notion of time increasing between them, such as $(t, 0)$, $(t, 1)$, $(t, 2)$, etc.[1] To convert a physical time T to a tag, for example to tag a sensor reading, one could simply assign the tag $(T, 0)$ to the sensor reading event.

To give a concrete example, the Lingua Franca (LF) coordination language [26] assumes that physical time measurements are Unix

[1] To understand the usefulness of such a model of time, consider Newton's cradle, the well-known toy with five steel balls hanging by strings. When one ball collides with the other four, its momentum is transferred through the three middle balls to the final ball. The three middle balls do not move. This transfer of momentum is usefully modeled as a sequence of events in superdense time.

time, which is a measurement of the number of nanoseconds that have elapsed since 00:00:00 UTC (Coordinated Universal Time) on 1 January 1970, the beginning of the Unix epoch, with adjustments made due to leap seconds. The LF runtime ensures that successive accesses to this physical time on any platform are strictly increasing and represents the result in a 64-bit signed integer (which will therefore overflow in the year 2262). LF uses superdense tags for events, where each tag consists of a 64-bit timestamp and a 32-bit unsigned superdense index. The timestamp of such events aligns with a local clock on a best-effort basis in that events will not be processed (by default) until a local physical clock matches or exceeds its timestamp. Similarly, asynchronous events injected into a running program, such as user interactions or sensor readings, are assigned timestamps based on an interrogation of a local clock.

Once an event is assigned a tag, it is handled in a such a way that every component in the system sees events in tag order. If two events have the same tag (they are logically simultaneous), then no component that watches for these events will see one as present and the other as absent at that tag. This policy makes it easy to build deterministic distributed systems that have a consistent view of system state at each logical time.

2.3 Consistency

In logical time, unlike time in physics, there is a well-defined notion of a globally shared instant. Using logical time, we can make statements like, "at (logical) time t, all components in the system agree that the value of a shared variable x is $x(t)$," a principle called "consistency." With care, we can design physical system realizations that adhere to this principle. A system implementation is consistent if for every t anywhere in the system, the local value $x(t)$ is the same as any $x(t)$ computed elsewhere in the system for the same logical time value t.

2.4 Implementing Consistent Systems

Now that we have a separation between logical and physical time and a notion of consistency, how do we build systems that correctly implement consistency? This ranges from easy to impossible, depending on the requirements of the system. Let us start with easy. Suppose that our system consists of two nodes, one of which updates x and the other of which updates y. Suppose further that we use a simple logical time model, like that of synchronous languages, where $t \in \mathbb{G} = \mathbb{N}$, the natural numbers, and $\mathcal{T}$ is some arbitrary monotonic function. In pseudo code, suppose the first component does this:

```
t = 0;
while(true) {
  x(t) = some update;
  send x(t) to the other component;
  wait for y(t) from the other component;
  if (x(t) + y(t) < 0) break;
  t = t + 1
}
```

Suppose the second component does the same thing, but with x replaced by y and vice versa. Such a system is consistent *even with no reference to physical time*. Suppose we add a connection to physical time as follows:

```
while(true) {
  ...
  t = t + 1
  wait for a local physical clock to reach t seconds
}
```

Now, we have established a relationship between logical time t and physical time. But that relationship is a bit subtle. In particular, suppose the two nodes have independent physical clocks, and no effort is made to synchronize them. Then the whole system will eventually align to the slower of the physical clocks. It will still remain consistent, in that for any $t \in \mathbb{N}$, the two nodes will agree on the values of $x(t)$ and $y(t)$, but they will disagree *by an arbitrarily large amount* on the discrepancy between t and their local measurement of physical time.

Consider now a more difficult case. Suppose that in each pass through the `while` loop above, each component may or *may not* send an update to the other component. Consistency could still be maintained by sending "null messages" whenever the component does not update its variable, but this could be inefficient, particularly if the updates are rare (as in most database applications).

A more clever solution might be to use some technique to synchronize the *physical* clocks [10, 17, 19, 24] then assume a bound E on the clock synchronization error and a bound L on the network latency. Then, instead of waiting for a message from the other component, each component could wait for its physical clock reading T to satisfy $T > t + E + L$. At that (physical) time, if it has not received an update from the other component, then it can assume that no such update is forthcoming and it can proceed. This is the essential principle behind PTIDES [35], and this principle is used in Google Spanner [8], a globally distributed database.

This implementation will be "correct," of course, only under the assumptions E and L. Consistency will only be maintained if the bounds E and L are not exceeded at run time. Fortunately, violations of these bounds are (eventually) detectable simply by including the tag t along with each message. If the first component receives a message with an update to y at logical time t_1, but its own local value of its variable t is bigger than t_1, then it knows that one of these assumptions has been violated (it is impossible to tell which one). In the case of Google Spanner, an update gets committed only after an acknowledgment has been received (more subtly, it uses a fault tolerant Paxos [18] consensus algorithm [8]). Consistency is maintained for traces consisting of all updates that are committed.

There is a long history with many other sophisticated methods for implementing consistent systems. In addition to the legacy from database systems [14], distributed simulation has also contributed a wealth of techniques [12]. Lingua Franca (see Section 3.4) realizes extensions of several of these techniques [1].

2.5 Consistency vs. Availability

Consistency is agreement on the value $x(t)$ of some shared state x at logical time t. Fundamentally, maintaining consistency comes at an unavoidable price in availability [5, 22]; specifically, there is a physical time delay that has to be imposed before a program can access the value of $x(t)$ in order to ensure consistency. As shown by Lee, et al. [22], this time delay is a linear function (in a max-plus algebra) of measurable delays due to networks, execution time, and clock synchronization. Moreover, this time delay can be

```
edge      = false -> (c and not pre(c));
edgecount = 0 -> if edge then pre(edgecount) + 1
                 else pre(edgecount);
```

Instant	0	1	2	3	4	5	6	7	8	9	10	11
c	f	t	t	f	t	f	t	f	t	t	f	t
pre(c)	?	f	t	t	f	t	f	t	f	t	t	f
edge	f	t	f	f	t	f	t	f	t	f	f	t
edgecount	0	1	1	1	2	2	3	3	4	4	4	5

Figure 1: A Lustre program fragment that counts false-to-true transitions on input `c` and its behavior on a stream.

reduced by relaxing consistency requirements; specifically, if we assert that component A's value of $x(t)$ at logical time t should agree with component B's value $x(t + \Delta)$ for some $\Delta > 0$, then a smaller physical time delay may be required to maintain this relaxed consistency. Lingua Franca (Section 3.4) supports explicit manipulations of this tradeoff between consistency and availability.

3 LOGICAL TIME IN LANGUAGES

A variety of languages and formalisms have emerged that embrace logical time. In this section, we review and compare a few of these.

3.1 The Synchronous Languages

Logical time in the synchronous languages Esterel, Lustre, and Signal [3] uses natural-number tags that count "instants" of computation, i.e., $\mathbb{G} = \mathbb{N}$. The relationship between tags and physical time, i.e., the functions $\mathcal{P}_c$ and $\mathcal{T}$, is implementation-dependent; most use periodic instants or the union of environmental inputs.

Each language describes systems that are concurrent, deterministic, and behave like single finite-state machines. Execution proceeds as a sequence of instants tagged $0, 1, 2, \ldots$. In each instant, the system examines its state and the inputs to compute its outputs for that instant and the next state. Computation can be logically instantaneous in that an input event may directly cause an output event with the same tag. Furthermore, programs can refer to the previous and next instant by observing the current state and controlling the next state, respectively. The Lustre program in Figure 1 illustrates this: the first line says that the value of the Boolean signal *edge* in the current instant is logical AND of the Boolean signal *c* in the current instant and its value from the previous instant.

Implementations typically take one of two approaches to tie logical and physical time. The simpler approach is periodic instants, as is done in synchronous digital logic circuits and periodically sampled control systems. Here, for any timestamp $g \in \mathbb{N}$, $d(\mathcal{T}(g+1), \mathcal{T}(g))$, where d is a metric on $\mathbb{T}$, is equal to the fixed clock period. Implementations employ a single periodic clock whose ticks are used to invoke a software "tick" function that reads environmental inputs, consults and updates the current state, and emits environmental outputs. As with synchronous digital logic, the periodic approach correctly implements the system provided the worst-case execution time of the "tick" function is less than the clock period. Note that in addition to synchronizing output events, the periodic approach also effectively constrains the environment to only deliver inputs on those instant boundaries.

The second, more general approach allows the environment to choose when instants occur, e.g., when there is an event on any environmental input. The periodic approach is a special case of this where all the environmental inputs have been forced to be synchronous with the clock. This approach allows the programmer to focus more on how to react to events rather than their relationship with the clock, but it does so at the expense of making it harder to determine whether the system can be implemented correctly. Specifically, this approach requires the reaction time to any input event is less than the time to the next event. Not only does this require knowledge of reaction time, it also requires knowledge of how quickly the environment will deliver inputs.

While the languages share a model of time and computation, they specify behavior differently. Esterel is an imperative language with the notion of multiple program counters; Lustre and Signal are both concurrent dataflow languages that provide subsampling in which portions of a system perceive only a subset of instants; Signal further provides supersampling where components may programmatically insert instants between those of another signal [4].

The languages' approaches to specifying and implementing intra-instant behavior also differ. A Lustre program [13] is a list of flow expressions (e.g., Figure 1); the Lustre compiler insists there be a static expression evaluation order that respects data dependencies. In particular, any self-referential cycle, such as that for `edgecount`, must be broken with a `pre`. Esterel's addition of control dependencies to Lustre-like data dependencies makes its behavior more difficult to compute and statically analyze, e.g., that a program will be causal (non-contradictory) in every instant. While the Esterel compiler also insists it can find a static order in which to execute statements in each instant, such orders may require interleaving statements and the order may even be state-dependent. Signal programs are potentially even harder to verify as their semantics are akin to constraint-solving over both values and timing.

3.2 The Sparse Synchronous Model

In a synchronous program implemented with a periodic clock, there is a tradeoff between timing precision and system complexity. Finer timing precision requires a shorter clock period, which demands less work be done in each instant. This forces designers to refactor higher-complexity tasks into operations across multiple instants, a difficult manual task similar to pipelining in digital logic designs.

The Sparse Synchronous Model (SSM) [9] avoids this tradeoff by adding computation across multiple instants to the synchronous languages' logical time model of natural-numbered tags ($\mathbb{G} = \mathbb{N}$), instantaneous execution, the ability to confront and deterministically resolve simultaneous events, and the distinction between intra- and inter-instant semantics.

However, unlike Lustre, Esterel, and Signal, SSM insists the instants are periodic, i.e., $d(\mathcal{T}(g+1), \mathcal{T}(g))$ is the fixed clock period). While an implementation knows its clock period, this is hidden from the user, who may only query, specify, and manipulate time as seconds. SSM does not expose the notion of the "next" or "previous" instant to the user. Under this policy, increasing the clock frequency should not affect program behavior.

As its name implies, SSM assumes computation is sparse: conceptually, a program only perceives a small fraction of all possible instants (specifically those in which an event arrives from the environment or is a result delivered by a multi-instant computation). Practically, however, the system is performing the work of long-running tasks during most of these logically idle instants.

Multi-cycle computation in SSM is specified by the *after* construct. When a statement such as "*after* 1 ms, $a \leftarrow b + c$" runs at some instant t, it captures the values of b and c at time t, starts the addition operation, schedules the value of a to be updated at logical time $t + 1$ ms, and terminates instantly (i.e., the current instant t is not updated) to allow other statements to be evaluated at the same (logical) instant. In traditional real-time task terminology, the addition task is released at time t and given a deadline of time $t + 1$ ms, although unlike in many real-time models, SSM delivers the result at exactly the (logical time) deadline and never earlier.

The SSM runtime system follows the logical time semantics and keeps itself synchronized to physical time. It relies on hardware timers to count instants with minimal software intervention. While the synchronous languages can express behavior like "wait until instant 1000," doing so requires the program to count the instants. By contrast, SSM has such delays in the form of the *after* primitive and implements it efficiently with an event queue that it keeps synchronized with physical time via hardware timers.

3.3 Logical Execution Time

The Logical Execution Time (LET) principle and its pioneering implementation in the Giotto language [11, 15, 16] depends heavily on the separation between logical time and physical time. Under this principle, the inputs to a software component are fixed at a logical time t and the outputs are produced at a later logical time $t + L$, where L is the logical execution time of the component, like that seen in the *after* primitive in SSM. A key advantage of the LET principle is that logical time and physical time can be more closely aligned because L accounts for the physical time that elapses during the computation performed by the component. This makes the interaction between software and its physical environment more controllable. On the other hand, too tight a binding between logical and physical time can make it difficult to take advantage of timing variability in software execution.

3.4 Lingua Franca

The Lingua Franca coordination language, which is meant to compose reactive segments of target code written in mainstream languages like C, Python, and Rust, embraces the concept of multiple timelines. The execution of an LF program involves the scheduling of tagged events and handling them in tag order. Events that enter the system asynchronously from the environment are pinned to a logical timeline using a tag derived from a reading of a physical clock. This happens via the scheduling of a "physical action." The scheduling of a "logical action," on the other hand, occurs in reaction to another event, and the tag of resulting is computed on the basis of the tag of the triggering event.

In LF, reactive code is encapsulated in "reactions," which are part of stateful components called "reactors" [25] which have ports that can be wired together using connections. Values produced on ports are logically instantaneous, meaning that they amount to events with the *same* tag as the events that triggered the reactions producing the outputs. Based on the connection topology between reactors and the signatures of their reactions (which specify which ports they have access to), the runtime system imposes scheduling constraints to ensure that no reaction executes until all of the ports it depends on have either settled on a final value or are known to be absent at that tag. Any two reactions that have no such dependencies between them may execute in parallel.

LF programs can be *federated*, in which case a program is split into multiple processes that can be run on distinct machines. The code generator synthesizes the communication and coordination so that, globally, every reactor sees events in logical time order [1].

In LF, by default, logical time "chases" physical time. When an event with the smallest tag g is to trigger reactions, the runtime system waits until a local clock c reads $\mathcal{P}_c(\tau) \geq \mathcal{T}(g)$. This gives a best-effort alignment of logical and physical times.

The `deadline` construct in LF gives another mechanism to relate logical and physical times. A reaction may be specified as follows:

```
reaction(triggers) -> effects {=
    target-language code
=} deadline (10ms) {=
    target-language code
=}
```

The second body of code will be invoked instead of the first if, when the reaction is triggered by an event with logical time g, a physical clock c reads $\mathcal{P}_c(\tau) > \mathcal{T}(g) + 10\text{ms}$. A deadline specifies an *upper bound* on the discrepancy between logical time and physical time.

By imposing logical time delays, a *lower bound* can be enforced on the difference between the logical time of an event and the physical time at which the event is reacted to in other parts of the system. For example, one can add an "`after`" clause to a connection between two reactors, which shifts the logical time at which an event is witnessed at the receiving end of the connection by a given amount of time. Adding logical delays breaks direct dependencies and hence relaxes scheduling constraints. This improves availability of the system at the cost of a measured loss in consistency [22].

3.5 Timed C

Timed C [31] is a C programming language extension that enables programming with time. The language consists of a small set of language primitives for specifying timed semantics, centered around the concept of *timing points*. For instance, the code fragment

```
while(true) {
  // Computation
  stp(10, 30, ms);
}
```

shows a *soft* timing point (`stp`) running in an infinite loop. The timing point specifies that the lower bound is 10 ms, and the upper bound is 30 ms. Conceptually, logical time only evolves at timing points and time does not advance when code is executed in-between timing points. In this example, logical time advances by 10 ms in each instance of `stp`, regardless of the computation time for the code between timing point invocations. Using real-time terminology, the lower bound of a timing point specifies the relative *arrival time*[2].

[2]The semantics for a clear distinction of arrival time and physical time was introduced by Natarajan *et al.* [33], compared to the original Timed C paper [31].

The upper bound dictates the *relative deadline*. In the case of a missed deadline for a soft timing point, the overshoot o is the difference between physical time and logical time, $o = \mathcal{P}_c(\tau) - \mathcal{T}(g_e)$, for some activation e of a timing point at a time instant τ. For a soft timing point, the deadline is not enforced, but can still be used during scheduling.

A firm timing point, `ftp`, is used when a missed deadline is not fatal, but the utility of the computation is zero. A firm timing point also has a lower and an upper bound, but where the deadline for the upper bound is enforced. That is, if the deadline is missed, the runtime jumps out from the computation and immediately proceeds to the next timing point. As a consequence, the programmer can handle missed deadlines. There is a special *critical section* construct to preserve data consistency.

The latest version of the semantics of Timed C [32] supports concurrent tasks that may communicate via FIFO channels or latest-value channels. There is currently no support for time-stamped values as part of the programming model, or incorporating logical time in a distributed setting, as in Lingua Franca. In contrast to synchronous programming languages, Timed C enables programs to reason about and react to disparities between logical time and physical time, such as deadline misses. Thus, like Lingua Franca, Timed C supports explicitly relaxing consistency to improve availability.

4 CONCLUSION

The notion of logical time, as distinct from physical time, is a critical element for engineering time-centric reactive systems. This notion appears in many related forms in synchronous languages, the logical execution time (LET) paradigm, and emerging programming languages and formalisms. It can give rigorous meaning to consistency and to control the practical timing of programs.

ACKNOWLEDGMENTS

The work in this paper was funded in part by the National Science Foundation (NSF), Award #CNS-2233769 (Consistency vs. Availability in Cyber-Physical Systems); the National Institutes of Health under grant RF1MH120034-01; the iCyPhy Research Center (Industrial Cyber-Physical Systems), supported by Denso, Siemens, and Toyota; the Swedish Foundation for Strategic Research (FFL15-0032); Digital Futures; and the Swedish Research Council (#2018-04329).

REFERENCES

[1] Soroush Bateni, Marten Lohstroh, Hou Seng Wong, Rohan Tabish, Hokeun Kim, Shaokai Lin, Christian Menard, Cong Liu, and Edward A. Lee. 2022. Xronos: Predictable Coordination for Safety-Critical Distributed Embedded Systems. *arXiv:2207.09555 [cs.DC]* (July 2022). hthttps://arxiv.org/abs/2207.09555

[2] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. 2012. The Fundamentals of Hybrid Systems Modelers. *J. Comput. System Sci.* 78, 3 (May 2012), 877–910.

[3] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The Synchronous Languages 12 Years Later. *Proc. IEEE* 91, 1 (Jan. 2003), 64–83.

[4] Albert Benveniste and Paul Le Guernic. 1990. Hybrid Dynamical Systems Theory and the SIGNAL Language. *IEEE Trans. Automat. Control* 35, 5 (May 1990), 535–546.

[5] Eric Brewer. 2017. *Spanner, TrueTime & the CAP Theorem.* Report. Google. https://storage.googleapis.com/pub-tools-public-publication-data/pdf/45855.pdf

[6] David Broman, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. 2015. Requirements for Hybrid Cosimulation Standards. In *Hybrid Systems: Computation and Control (HSCC).*

[7] A. Cataldo, E. Lee, Xiaojun Liu, E. Matsikoudis, and Haiyang Zheng. 2006. A constructive fixed-point theorem and the feedback semantics of timed systems. In *8th International Workshop on Discrete Event Systems.* 27–32.

[8] James C. Corbett et al. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI.*

[9] Stephen A. Edwards and John Hui. 2020. The Sparse Synchronous Model. In *Forum on Specification and Design Languages (FDL).* Kiel, Germany.

[10] John C. Eidson. 2006. *Measurement, Control, and Communication Using IEEE 1588.* Springer.

[11] Rolf Ernst, Stefan Kuntz, Sophie Quinton, and Martin Simons. 2018. The Logical Execution Time Paradigm: New Perspectives for Multicore Systems (Dagstuhl Seminar 18092). *Dagstuhl Reports* 8 (2018), 122–149. https://doi.org/10.4230/DagRep.8.2.122

[12] Richard Fujimoto. 2000. *Parallel and Distributed Simulation Systems.* John Wiley and Sons, Hoboken, NJ, USA.

[13] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The Synchronous Data Flow Programming Language LUSTRE. *Proc. IEEE* 79, 9 (Sept. 1991), 1305–1320.

[14] Pat Helland. 2016. Standing on Distributed Shoulders of Giants. *Commun. ACM* 59, 6 (2016), 58–61.

[15] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. 2003. Giotto: A Time-Triggered Language for Embedded Programming. *Proceedings of IEEE* 91, 1 (January 2003), 84–99.

[16] Christoph M. Kirsch and Ana Sokolova. 2012. The Logical Execution Time Paradigm. In *Advances in Real-Time Systems*, S. Chakraborty and J. Eberspächer (Eds.). Springer-Verlag, Berlin Heidelberg, 103–120.

[17] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical Physical Clocks. In *Principles of Distributed Systems (OPODIS)*, Aguilera M.K., Querzoni L., and Shapiro M. (Eds.), Vol. LNCS 8878. Springer.

[18] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

[19] Leslie Lamport and P. M. Melliar-Smith. 1984. Byzantine clock synchronization. In *Proceedings of the third annual ACM symposium on Principles of distributed computing.* ACM Press, Vancouver, British Columbia, Canada, 68–74.

[20] Edward Ashford Lee. 2017. *Plato and the Nerd — The Creative Partnership of Humans and Technology.* MIT Press.

[21] Edward A. Lee. 2021. Determinism. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 5 (July 2021), 1–34.

[22] Edward A. Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. 2023. Trading Off Consistency and Availability in Tiered Heterogeneous Distributed Systems. *Intelligent Computing* 2, 13 (February 15 2023), 1–23.

[23] Edward A. Lee and Alberto Sangiovanni-Vincentelli. 1998. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems* 17, 12 (December 1998), 1217–1229.

[24] Barbara Liskov. 1993. Practical uses of synchronized clocks in distributed systems. *Distributed Computing* 6 (1993), 211–219.

[25] Marten Lohstroh, Íñigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. 2019. Reactors: A Deterministic Model for Composable Reactive Systems. In *8th International Workshop on Model-Based Design of Cyber Physical Systems (CyPhy'19)*, Vol. LNCS 11971. Springer-Verlag, 27.

[26] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a Lingua Franca for Deterministic Concurrent Systems. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 4 (May 2021), Article 36.

[27] Oded Maler, Zohar Manna, and Amir Pnueli. 1992. From Timed to Hybrid Systems. In *Real-Time: Theory and Practice, REX Workshop.* Springer-Verlag, 447–484.

[28] Zohar Manna and Amir Pnueli. 1993. Verifying Hybrid Systems. In *Hybrid Systems*, Vol. LNCS 736. 4–35.

[29] D. L Mills. 2003. A brief history of NTP time: confessions of an Internet timekeeper. *ACM Computer Communications Review* 33 (April 2003).

[30] Richard A. Muller. 2016. *Now — The Physics of Time.* W. W. Norton and Company.

[31] Saranya Natarajan and David Broman. 2018. Timed C: An Extension to the C Programming Language for Real-Time Systems. In *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).*

[32] Saranya Natarajan and David Broman. 2020. Temporal Property-Based Testing of a Timed C Compiler using Time-Flow Graph Semantics. In *Forum for Specification and Design Languages (FDL).* IEEE, 1–8.

[33] Saranya Natarajan, Mitra Nasri, David Broman, Björn B Brandenburg, and Geoffrey Nelissen. 2019. From code to weakly hard constraints: A pragmatic end-to-end toolchain for Timed C. In *Proccedings of the IEEE Real-Time Systems Symposium (RTSS).* IEEE, 167–180.

[34] Carlo Rovelli. 2018. *The Order of Time.* Riverhead Books, New York.

[35] Yang Zhao, Edward A. Lee, and Jie Liu. 2007. A Programming Model for Time-Synchronized Distributed Real-Time Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS).* IEEE, 259 – 268.