



# Timetide: A Programming Model for Logically Synchronous Distributed Systems

LOGAN KENWRIGHT, Electrical, Computer, and Software Engineering, University of Auckland, Auckland, New Zealand

PARTHA ROOP, Electrical, Computer, and Software Engineering, University of Auckland, Auckland, Auckland, New Zealand

NATHAN ALLEN, Engineering, Computer and Mathematical Sciences, Auckland University of Technology, Auckland, New Zealand

CALIN CASCAVAL, Google DeepMind, Auckland, New Zealand

AVINASH MALIK, Electrical, Computer, and Software Engineering, University of Auckland, Auckland, Auckland, New Zealand

Massive strides in deterministic models have been made using synchronous languages. They are mainly focused on centralised applications, as the traditional approach is to compile away the concurrency. Time triggered languages such as Giotto and Lingua Franca are suitable for distribution albeit that they rely on physical clock synchronisation, which is both expensive and may suffer from scalability. Hence, deterministic programming of distributed systems remains challenging.

We address the challenges of deterministic distribution by developing a novel multiclock semantics of synchronous programs. The developed semantics is amenable to seamless distribution. Moreover, our programming model, Timetide, alleviates the need for physical clock synchronisation by building on the recently proposed *logical synchrony* model for distributed systems. We discuss the important aspects of distributing computation, such as network communication delays, and explore the formal verification of Timetide programs. To the best of our knowledge, Timetide is the first multiclock synchronous language that is both amenable to distribution and formal verification without the need for physical clock synchronisation or clock gating.

CCS Concepts: • **Computing methodologies** → **Distributed programming languages**; • **Theory of computation** → *Models of computation; Formalisms.*

Additional Key Words and Phrases: Logical Synchrony, Synchronous Programming Languages, Distributed Systems, bittide

---

Authors' Contact Information: Logan Kenwright, Electrical, Computer, and Software Engineering, University of Auckland, Auckland, New Zealand; e-mail: logan.kenwright@auckland.ac.nz; Partha Roop, Electrical, Computer, and Software Engineering, University of Auckland, Auckland, Auckland, New Zealand; e-mail: p.roop@auckland.ac.nz; Nathan Allen, Engineering, Computer and Mathematical Sciences, Auckland University of Technology, Auckland, New Zealand; e-mail: nathan.allen@aut.ac.nz; Calin Cascaval, Google DeepMind, Auckland, New Zealand; e-mail: cascaval@google.com; Avinash Malik, Electrical, Computer, and Software Engineering, University of Auckland, Auckland, Auckland, New Zealand; e-mail: avinash.malik@auckland.ac.nz.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 1558-3465/2025/9-ART

<https://doi.org/10.1145/3763794>

## 1 Introduction

Deterministic programming of distributed systems remains challenging [26] in spite of decades of formal models for such systems, such as Kahn Process Networks (KPNs) [15]. Despite the advantages of deterministic execution, the vast majority of distributed systems are asynchronous and thus non-deterministic. For example, the widely used actor-based models are inherently non-deterministic [30]. Non-deterministic concurrency, while being a desirable feature for specification [20, 32], is hard to verify and debug. In contrast, synchronous languages [5] alleviate this by “compiling away” the concurrency, eliminating any run-time uncertainty. However, the distribution of these programs remains challenging. While many approaches have been studied [16], they are not scalable in general. Hence, achieving both good performance and determinism is an unsolved problem, as most approaches rely on expensive physical clock synchronisation.

The need to synchronise distributed components in a system raises at least two major challenges. One is achieving maximum throughput — when a system has dependencies between components that must be synchronised on, these cannot execute freely. Rather, they must wait for any upstream data to arrive. Consequently, systems which execute in lock-step, such as Loosely Time-Triggered Architectures (LTTAs) [37], experience worsening throughput proportional to the transmission delay between machines. The second challenge is balancing the requirements of consistency and availability [27]. In a distributed system, a designer must choose between the ability to retrieve the true value of a shared variable (consistency) and the amount of time it takes a system to respond to a request (availability). Typically, synchronous systems are completely consistent, which comes at the cost of availability. By specifying delays between tasks, we propose to relax the consistency requirement, and thus improve the availability of the system. The recently proposed *logical synchrony* model [24] describes an abstract model for distributed systems where delays are exactly specified, and forms the basis for the Timetide language presented in this work.

### 1.1 Logical Synchrony

While there exist many approaches methods [11, 28] that exchange timestamps based on protocols such as Precision Time Protocol (PTP) [1], these are expensive to build for high precision accuracy. *Logical synchrony* [24] is a viable alternative to physical clock synchronization. Logical synchrony provides a shared notion of time sufficient for reasoning about causality without requiring a shared system-wide clock. Applications running on the system use a local logical clock and derived knowledge of their peers’ logical clocks to coordinate their actions, which replaces the need to reference physical time.

Logical Synchrony Networks (LSNs) [21] are a graph-based Model of Computation (MoC) based on logical synchrony, which capture computational machines as the nodes of the graph and uni-directional links between them for communication. Each link has an invariant logical delay between production and consumption of tokens. This ensures communication-level determinism, as proven in [21], by establishing that the execution forms a Complete Partial Order. We recap the definition of LSN from [21] for completeness.

**Definition 1.** An LSN is defined as a tuple  $\mathcal{L} = (G, \Theta, \lambda)$ , where:

- $G = (V, E)$  is a directed graph of  $V$  vertices (the distributed machines  $\mathcal{M}$ ) and  $E \subseteq V \times V$  edges connecting them (the communication links), with  $\forall_{(v1, v2) \in E} : v1 \neq v2$ ,
- $\Theta$  represents the set of local clock valuations for each node, such that the valuation of a clock  $\theta_i \in \Theta$  for machine  $\mathcal{M}_i \in V$  at a given time  $t$  is denoted as  $\theta_i(t) \in \mathbb{N}$ , and
- $\lambda : E \rightarrow \mathbb{Z}$  captures the communication delay for each link in the network. For simplicity, the delay between  $\mathcal{M}_i$  and  $\mathcal{M}_j$  is denoted as  $\lambda_{i \rightarrow j} = \lambda((\mathcal{M}_i, \mathcal{M}_j))$ .

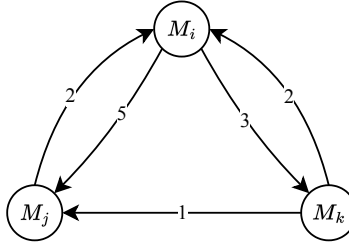


Fig. 1. An example of a Logical Synchrony Network

Figure 1 shows an example of an LSN, where the numbers on each edge correspond to the logical delays  $\lambda$ . At each logical tick for a single node, a unit of data called a *frame* is consumed on each incoming edge of the graph, and a frame is produced on each outgoing edge. A send event at a node is delayed by a fixed logical delay before it is consumed by a receiving node. This relationship is captured in Equation (1) as follows:

$$\theta_j(t_{receive}) = \theta_i(t_{send}) + \lambda_{i \rightarrow j} \quad (1)$$

where  $\theta_i(t_{send})$  is the logical clock value of a sender at the time it sends,  $\lambda_{i \rightarrow j}$  is the fixed logical latency between the sender and receiver, and  $\theta_j(t_{receive})$  is the logical clock value of the receiver at the time it receives the message. Notably, the specific values of  $t_{send}$  and  $t_{receive}$  are not important, as the logical delay is fixed.

Logical synchrony may be implemented at the system level using the approach given by the *bittide* [25] protocol, where nodes synchronize by monitoring the rate of communication with their neighbors without requiring a global clock. Alternatively, logical synchrony has also been implemented using Kahn-like token pushing networks [21]. In both approaches, when viewed from the outside, the shared logical time is fully disconnected from physical wall-clock time, meaning that logical time steps can vary in physical duration. There is therefore no requirement for systems to be completely synchronised in their logical clocks at any physical instant from the point of view of a hypothetical omniscient observer, as long as the logical delay invariance can always be maintained. The logical synchrony model only describes the communication behaviour and does not provide a programming model for the tasks running on the nodes. We will elaborate on this in the following sections.

## 1.2 Logical Execution Time Task Model

The Logical Execution Time (LET) model is a programming abstraction which describes networks of communicating tasks. LET is commonly used for modelling timed concurrent systems as it exhibits timing-determinism. Such systems are typically cyber-physical systems, composed of one or more processing cores driven by a single clock. A LET task, shown in Figure 2, is expressed in languages like Giotto [22]. Each task has a *period* describing how often a task begins execution, a *duration* which describes how long between the time a task begins until it emits an output, an *initial offset* describing the time between the start of synchronous clock and the first release of the task for execution, and a *period offset* describing how far into the repetition cycle the task body begins.

As an alternative, the LET model may be combined with synchronous programming in the Synchronous Logical Execution Time (sLET) model [34]. Rather than synchronise on physical timestamps, sLET tasks synchronise on named clocks, gaining the benefits of synchronous programming, but cannot express large transmission delays. The System-Level Logical Execution Time (SL-LET) model [13] explicitly specifies

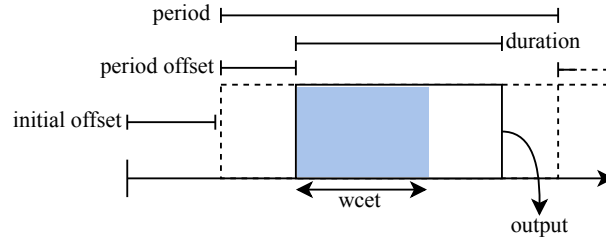


Fig. 2. Structure of a (typical) LET task

transmission delays, allowing for the modelling of systems with non-negligible latency. Unlike the sLET model, the SL-LET model does not make use of logical time for verification.

### 1.3 Timetide: deterministic distributed programming

Currently, there are no programming models which combine the purely synchronous approach of sLET with the explicit communication delays of SL-LET. Existing distributed programming languages generally ignore latency and require that a designer chooses between non-determinism or expensive physical clock synchronisation. Timetide is proposed to unify the sLET and SL-LET approaches by allowing the specification of logical communication delays between tasks. This allows the user to form powerful execution pipelines.

Thus, for the first time, we can express high-performance distributed systems with deterministic execution and no physical clock synchronisation in the presence of non-negligible transmission latencies. By leveraging LSNs for the underlying execution model, we separate the synchronisation layer from the application layer and allow Timetide programs to execute over a wide range of synchronisation methods.

The main contributions of Timetide are as follows:

- (1) We introduce the Timetide language for deterministic distributed systems. Timetide is the first language based on logical synchrony [24].
- (2) Timetide treats communication delays as first-class citizens, a feature absent in other languages for distributed systems.
- (3) We introduce LSN-compatibility to specify architectures that can deploy Timetide programs, executing either as centralised or distributed applications.
- (4) We present the formal semantics for Timetide. This facilitates deterministic distribution without the need for physical clock synchronisation.

The paper is structured as follows. Firstly, we introduce a motivating example in Section 2. Subsequently, we present the constructs of Timetide in Section 3.1. In Section 3.3, we propose a set of statements, we term the *kernel language*, using which we can express all other language constructs. We use these statements to develop the operational semantics of Timetide and formalise the key properties of the language. Subsequently, in Section 4, we formally prove the determinism of this language. In Section 5 we develop a simple source to source translation to the well-known Esterel language, to leverage existing tools for compilation and verification. In Section 5.3, we illustrate how Timetide programs are implementable over LSN-compatible architectures. In Section 6, we compare and benchmark Timetide against the deterministic language *Lingua Franca*. The paper finishes by comparing this approach to the related work and making concluding remarks, including the scope for future developments of Timetide.

## 2 Motivating Example

We motivate Timetide using an application modelling a financial trading system, where logical time, not physical time, is used to arbitrate trades, ensuring that all traders get a fair chance to participate. The system consists of a single *Exchange*, and an arbitrary number of *Traders*. Both the traders and the exchange execute periodic tasks. Periodically, the exchange performs an execution where it checks whether any trade orders have arrived, matches the buy/sell orders, and sends order confirmations back to the traders, as well as the price spread showing the best buy/sell prices. The price spread is the best bid and ask prices, along with the quantity available at each price, as shown in Table 1.

Table 1. A price spread of the three best bid and ask prices

Side	Price	Quantity
Buy	100	97
Buy	99	32
Buy	98	121
Sell	101	88
Sell	102	42
Sell	103	79

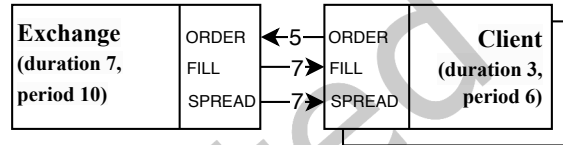


Fig. 3. A financial trading system modelled in Timetide with logical delays marked on edges

The traders similarly perform a periodic task, albeit at a different rate. Each cycle a trader reads any responses to their previous orders and the latest price spread. If a new spread has been received since the last cycle the trader may decide to place a new order. Each task takes a number of ticks to complete before emitting a value. This program is visualised in Figure 3, showing the chosen durations, periods, and transmission delays.

In a practical scenario the values for the periods, durations, and delays will be chosen based on the system specifications. For example, in a trading setting, enforcing the delay between each of the traders and the exchange to be equal ensures a fair playing field. Alternatively, the exchange may choose to create tiers of service that traders can subscribe to.

## 3 The Timetide language

The Timetide language is designed to support the logical synchrony model abstractions: a network of communicating tasks, synchronized on a logical clock. In Timetide the fundamental unit of modularity is the *module*, which can either perform a periodic *task* or instantiate other modules. Thus, any module may be chosen to be the top level. Tasks follow the LET model: they have periodic release times (ticks at which they begin a cycle of execution) and a duration (or deadline, by which computation must end). Outputs are ready to send once the deadline has passed. Tasks communicate using *channels* with fixed logical delays. Channels are read from and written to on each clock tick, however tasks only sample the most recent value read from the channel at their release time, which may cause interim values to be missed. Yet, because deadlines and delays are exact, programs can be analyzed to detect such cases. Similarly, outputs are only updated at the end of a task duration.

### 3.1 Syntax

The Timetide syntax is inspired by Esterel, since it provides a solid foundation for expressing synchrony. Moreover, as we will show later, we can translate Timetide to Esterel and take advantage of its verification tools. The list of Timetide statements is shown in Table 2.

Table 2. Timetide statements

Statement	Meaning
<code>module <i>m</i>: ... end module</code>	declare a module
<code>input [const] &lt;name&gt; : <i>type</i></code>	declare an input port to a module
<code>output &lt;name&gt; : <i>type</i></code>	declare an output port to a module
<code>channel <i>ch</i> : [<i>type</i>] delay <math>\delta</math></code>	declare a named channel <i>ch</i> of <i>type</i> with delay $\delta$
<code><i>t</i>&lt;&gt;<i>u</i></code>	concurrently execute program statements <i>t</i> and <i>u</i>
<code>run &lt;module&gt; [&lt;channel&gt;/&lt;port&gt;, ...]</code>	instantiate a module with channel bindings
<code>foreach <i>i</i> in &lt;const or num&gt; {<i>t</i>}</code>	execute <i>t</i> in sequence for each value of <i>i</i>
<code>pareach <i>i</i> in &lt;const or num&gt; {<i>t</i>}</code>	execute <i>t</i> in a parallel thread for each value of <i>i</i>
<code>var <i>v</i> : <i>type</i> [= &lt;initial&gt;] in <i>t</i> end</code>	declare a local variable <i>v</i> of <i>type</i> scoped to <i>t</i>
<code>const <i>c</i> : <i>type</i> = &lt;value&gt;</code>	declare a compile-time constant
<code><i>t</i>; <i>u</i></code>	run <i>t</i> , and then <i>u</i> in sequence
<code>task(period=<i>p</i>,duration=<i>d</i>,offset=<i>o</i>): <i>t</i> end</code>	run <i>t</i> every <i>p</i> ticks, for <i>d</i> ticks, starting at <i>o</i> ticks
<code>[weak?] abort <i>t</i> when [immediate?] <i>expr</i></code>	(Weak) abort the body when <i>expr</i> becomes true
<code><i>v</i> = <i>f</i>(...)</code>	assign <i>v</i> with the expression <i>f</i>
<code>if <i>c</i>(...) <i>t</i> else <i>u</i></code>	run <i>t</i> if condition <i>c</i> ; otherwise <i>u</i>
<code>&lt;expr&gt;</code>	evaluate an expression
<code>send <i>ch</i>(&lt;<i>expr</i></code>	send a value along channel <i>ch</i>
<code>fresh(<i>ch</i>)</code>	true if the value in <i>ch</i> has not been sampled yet
<code>+, -, /, *, &gt;, &lt;, &lt;=, &gt;=, !</code>	arithmetic operators
<code>and, or</code>	boolean operators
<code>&lt;ident&gt;</code>	read a variable, constant, or sampled channel

We illustrate the syntax using an implementation of the financial trading example from Section 2. Figure 4 shows the top level Timetide specification of the financial trading system example which instantiates a single exchange *center* and two *traders*. Channels are specified with their types and logical delays. Each input and output port of a module is mapped by the top level to the fixed-delay channels in the network. Threads are specified through parallel arms of the parallel <> operator. Threads will run concurrently, co-located or distributed based on a schedule defined by the mapping to an actual architecture. Channel routing is automatically inferred based on their usage in the module, in this case by the `run` statement which instantiates a module template.

Two types of basic iterators are supported in Timetide: `foreach`, which declares a sequential loop, and `pareach`, which declares a (distributed) parallel loop. Both of these are simple substitution macros which unravel into a sequence (`t;u;v`) and parallel blocks (`t<>u<>v`) respectively. Only a compile-time constant is allowed as the iterator bound, as all threads in Timetide have static lifetime.

Figure 5 shows the declaration of the module for the exchange Center. Section 3.1 assigns a module name. As we've seen, the module name is then used by the `run` statement to spawn work. Each module may have input and output declarations (section 3.1 - 5), which may be endpoints of a channel or a compile-time constant which is passed in. The body of the module consists of statements that are evaluated sequentially. The `var` statement on section 3.1 declares a scoped variable block. Variables are mutable values which are local to their enclosing scope and cannot be shared between threads. The `task`

statement (section 3.1) declares a repeating periodic task (similar to a typical LET task), specifying its period, duration, and optional offset from the beginning of the period to start of work. Section 3.1 shows a variable assignment, and the current value of the `order` channel being read. The `send` statement (section 3.1) writes to a buffer in data memory for the associated channel such that at the end of the task body the value is sent to the tail of channel. We also show the Timetide specifications for the trader component in Figure 6.

```

1 module toplevel:
2   const TRADERS : int = 2;
3   channel orders : Order[TRADERS] delay 5;
4   channel fills : MatchedOrders[TRADERS] delay 7;
5   channel spreads : Spread[TRADERS] delay 7;
6   {
7     run Center(TRADERS/TRADERS, orders/orders, fills/fills, spreads/spreads);
8   }
9   <>
10  pareach i in TRADERS {
11    run Trader(spreads[i]/price_spread, orders[i]/order, fills[i]/fill, i/id);
12  }
13 end module;

```

Fig. 4. The Timetide code of the top level module

```

1 module Center:
2   input const TRADERS : int;
3   input orders : Order[TRADERS];
4   output fills : MatchedOrders[TRADERS];
5   output spreads : Spread[TRADERS];
6   var orderbook : OrderBook = create_orderbook(4) in
7     task(period=10, duration=7):
8       foreach i in TRADERS {
9         if (fresh(order[i])) {
10          orderbook = insert_order(orderbook, order[0]);
11        }
12      }
13   var matched_orders : MatchedOrders = run_matching(orderbook) in
14     send fill(matched_orders);
15   end var;
16   var price_spread : Spread = get_spread(orderbook) in
17     send fill(price_spread);
18   end var;
19 end task;
20 end var;
21 end module;

```

Fig. 5. The Timetide code of the controller module

Parallel instantiations of modules communicate over channels — point-to-point communication media that behave as First In First Out (FIFO) queues. At each local tick on the sender’s clock, a value is sent on the channel, and at each local tick on the receiver’s clock, a value is read from the channel. The value sent on the channel is specified by the most recent `send` statement used within the task body. If this tick does not align with the end of a task, an empty value will be sent instead. These channels are logically synchronous, meaning that they have a fixed logical delay specified. Let  $\mathcal{V}$  be the set of values that can

be written to a channel. Then we can define the input and output streams of a channel  $q$  as follows:

$$q_{\text{in}} : \mathbb{N} \rightarrow \mathcal{V}, q_{\text{out}} : \mathbb{N} \rightarrow \mathcal{V} \quad (2)$$

where  $q_{\text{out}}(n)$  represents the value pushed onto the channel at sender clock  $n$ , and  $q_{\text{in}}(m)$  represents the value read from the head of the channel at receive clock  $m$ . The logical delay  $\delta_{i \rightarrow j} \in \mathbb{N}$  relates the input and output streams of a channel from node  $i$  to node  $j$  as follows:

$$q_{\text{in}}(m) = q_{\text{out}}(m - \delta_{i \rightarrow j}), \quad \text{for all } m \geq \delta_{i \rightarrow j} \quad (3)$$

For  $m < \delta_{i \rightarrow j}$ , the channel will contain empty data unless initial values are specified explicitly. For modules which don't have a channel dependency,  $\delta_{i \rightarrow j} = \perp$

```

1 module Trader:
2   input price_spread : Spread;
3   output order : Order;
4   input fill : MatchedOrders;
5   input id : integer;
6   var balance : float = 1000.0 in
7     var outstanding : OrderList = OrderList_create(id) in
8     task(period=6, duration=3):
9       if (fresh(fill)) {
10        var change : float = update_orders(fill, outstanding, id) in
11        balance = balance + change;
12        end var;
13      }
14      if (fresh(price_spread)) {
15        var new_order : Order = make_decision(price_spread, outstanding, balance, id) in
16        if (should_do_trade(new_order)) {
17          send order(new_order);
18        }
19        end var;
20      }
21    end task;
22  end var;
23  end var;
24 end module;

```

Fig. 6. The Timetide code of the trader module

**3.1.1 Program Structure.** A Timetide program consists of a network of logically synchronous threads. Different threads are distinguished by the use of the parallel operator  $\langle \rangle$ . The `module`, `input`, and `output` statements are syntactic sugar for the modular instantiation of blocks of statements. Figure 7 shows the structure of a program with three distinct threads, containing the terms  $w$ ,  $u$ , and  $v$  respectively.

```

1 {w} <> {u <> v};

```

Fig. 7. A hierarchy of logically synchronous threads

Each thread  $t$  has an associated logical clock  $\theta_t \in \Theta$ . Threads may only communicate using named channels with a logical delay, as shared state is not possible between potentially distributed threads. Channels which are passed into a nested thread from an enclosing thread may not be used in any other thread, including the enclosing thread — scoping of parallel threads is entirely syntactic for modularity

purposes, but does not carry any special meaning from a semantic standpoint. A term within a thread is an arbitrary sequence of statements, which may include other thread declarations or a periodic task.

**3.1.2 Task Structure.** Consider the task shown in Figure 8, with period 4, offset 1, and duration 2, and its corresponding translation to semantic constructs.

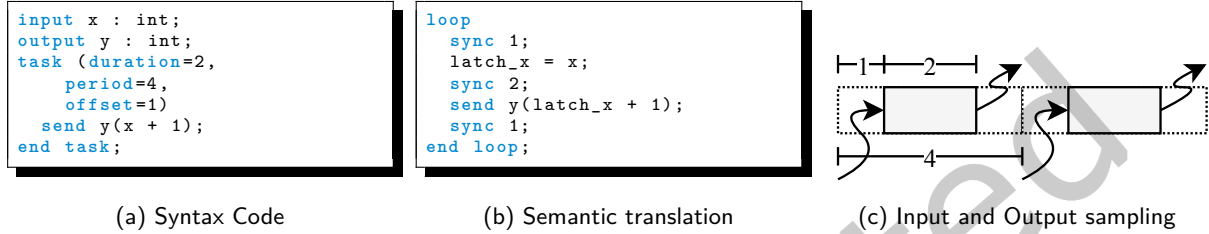


Fig. 8. A task with period 4, offset 1, and duration 2, showing input sampling and output production

The task syntax is converted semantically to a combination of various semantic constructs.

- An infinite loop construct is used to describe the repeating nature of the task
- A `sync` statement is inserted at the beginning of the loop to model the offset. The `sync` statement progresses the logical clock by  $d$ , pops  $d$  values from the head of each channel, and enqueues  $d$  values from the data memory onto the tail of each channel
- A generated variable `latch_x` is used to store the value of the input variable `x` at the sampling time of the task, according to LET semantics.
- A `sync` statement is inserted after the input sampling to progress the logical clock by the duration of the task
- Following the duration, the body of the task is executed in zero logical time.
- Finally, a `sync` statement is inserted at the end of the task to satisfy any extra time between the  $\text{duration} + \text{offset}$  and the period.

In Figure 8, we can also see that there is no guarantee that data will be generated every logical tick. Instead, it depends on the *period* of the task. The `fresh` operator is used to determine whether new data has arrived on the receiving end of a channel.

### 3.2 Pipelining

A task may have duration longer than its period, in which case the task is pipelined. Pipelining is achieved by inserting additional parallel threads that execute the task body at the correct rate. The number of parallel threads spawned can be determined by the ratio of the Least Common Multiple (LCM) of the period and duration, as shown in Equation (4).

$$\text{number of threads} = \frac{\text{lcm}(d, p)}{p} \quad (4)$$

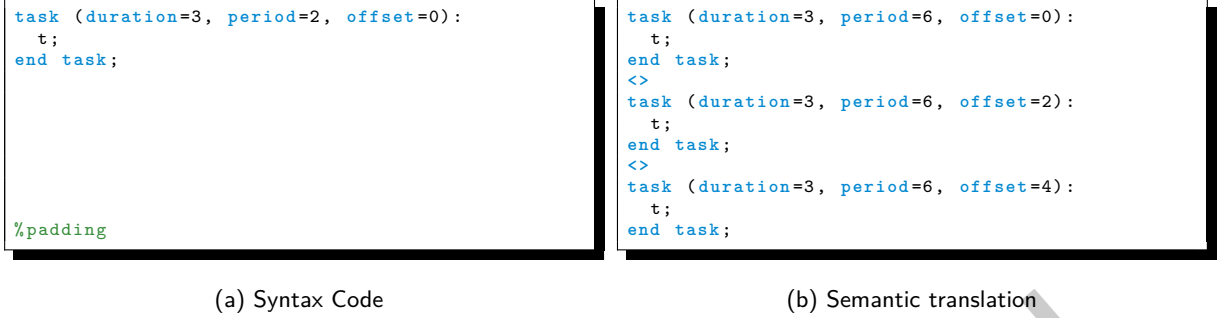


Fig. 9. A sugared Timetide task, and its pure translation

The translation to parallel threads is shown in Figure 9, and the associated timing behaviour for the task is shown in Figure 10. A pipeline is automatically formed by the compiler such that the difference in release times of the tasks is equal to the period of the original task.

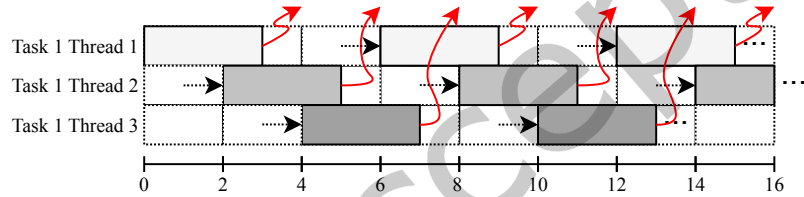


Fig. 10. Timing of a task with period 2 and duration 3. Dashed input arrows to each task cycle show is multiplexed from the same input, red arrows show output production

### 3.3 Semantics

The formal semantics for the kernel statements are presented as program transitions using Structural Operational Semantics (SOS) rules [33] of the following form:

$$\{t, D, Q, \Theta\} \rightarrow \{t', D', Q', \Theta'\}$$

where

- $t$  is the sequence of statements that belong to a thread,
- $D$  is the set of values of data variables associated with  $t$  before the transition,
- $Q$  is a global mapping from a channel label to its associated queue with each  $q$  in  $Q$  having an associated data variable  $fresh_q$  which is true if the most recent non-empty value at the head of channel has not been read yet,
- $\Theta$  is the set of clocks before the transition,
- $\Theta'$  is the set of clocks after the transition,
- $t'$  is the residual (remaining kernel statements) of  $t$  after the transition,
- $D'$  is the set of values of data variables associated with  $t$  after the transition, and
- $Q'$  maps a channel label to its associated queue after the transition

Each parallel thread (e.g.  $t \langle \rangle \dots$ ) in Timetide has an associated clock  $\theta_t$ . Because most statements work within the scope of a single clock, we use the shorthand  $\{t, D, Q\} \xrightarrow[\theta]{\theta'} \{t', D', Q'\}$  to denote that the transition only modifies the associated clock of the thread in which  $t$  is executing. Transition predicates are expressed through reduction rules of the form

$$\frac{\langle \text{pred} \rangle}{\{t, D, Q\} \xrightarrow[\theta]{\theta'} \{t', D', Q'\}}$$

where the  $\langle \text{pred} \rangle$  must hold in order for the transition below the bar to happen. When no such dependency exists, the bar is omitted for simplicity. Table 3 provides an overview of the SOS rules for each of the kernel statements of the Timetide language.

**3.3.1 Channel and Data Operations.** Every *sync* statement inherently performs a send and a receive operation along each channel. During a send operation, the channel reads from a mailbox in the data store  $D$  that contains the value to be sent at the next *sync*. A programmer can write to this unique location using the **send** statement. Rule **SEND** describes the sending of a value over a channel. The **send** statement is rewritten into a **nothing** statement after the completion of the transition. Similarly, the value of a channel is read from a unique memory location in  $D$  where it was written to by the dequeue operation of the most recent *sync* statement. For each non-empty value popped from each queue in  $Q$ , the value is updated in the datastore and the freshness flag is set to true. The value popped is the value written to the channel buffer exactly  $\delta$  ticks ago, with respect to the sender clock.

The **read** statement returns the most recent non-empty value popped during a *sync* statement, and is rewritten into its resulting value expression (Rule **READ**).

The **fresh** statement checks if the current value in a channel has been read previously, and returns a boolean value **true** if it has not been read, or **false** otherwise (Rule **FRESH**). The **fresh** statement is only used in expressions, and is rewritten into a **nothing** statement.

**3.3.2 Control Flow.** Rule **SEQ1** expresses the fact that the sequence does not finish, if its left branch,  $t$ , does not. If the left branch advances, so does the sequence (Rule **SEQ1**). Otherwise, control will be immediately transferred to the right branch.  $u$ , when  $t$  finishes (Rule **SEQ2**).

**3.3.3 Parallelism.** The clock associated with each program term is used to synchronize its behaviour with respect to others in the system. Synchronisations are required when there exists a unidirectional channel between two terms, forming a dependency. Channel delays are exact in Timetide, meaning a receiving thread will not be able to progress its logical clock past  $\theta_{rcv}$  until the sending term has progressed to  $\theta_{snd} = \theta_{rcv} - \delta$ , where  $\delta$  is the channel delay (Rule **DPAR1**). If term  $t$  has reduced to nothing, then only term  $u$  may progress. In this case, the clock guard  $\theta'_t - \theta_u \leq \delta_{u \rightarrow t}$  is not required (Rule **DPAR2**).

**3.3.4 Preemption.** Timetide provides a mechanism for preemption using the **abort** statement. Timetide adopts the PRET-C [2] approach of encoding aborts, where **checkabort** statements are placed into the body at compile time. For strong aborts, a single **checkabort** statement immediately follows each *sync* statement in the term. For every weak abort, a **checkabort** statement is inserted directly before each *sync* statement in the term, except for the first. If an *immediate* qualifier is added, a **checkabort** is inserted at the first line of the term for a strong abort, or immediately before the first *sync* for a weak abort. When aborts are nested, priority is given to the outermost abort statement through the placement of the **checkabort** statements. Rule **ABORT1** captures the case where the exit condition is not met, and so the term progresses. Conversely, Rule **ABORT2** captures the case where the exit condition is met and the

Table 3. Structural Operational Semantics Rules for Timetide Kernel statements

<p><b>NOTHING:</b></p> $\{\text{nothing}, D, Q\} \xrightarrow{\theta} \{\text{nothing}, D, Q\}$ <p><b>VARDECL:</b></p> $\{\text{var } x, D, Q\} \xrightarrow{\theta} \{\text{nothing}, D', Q\}$ <p>where <math>D' = D \cup \{x = \perp\}</math></p> <p><b>CHANDECL:</b></p> $\{\text{channel } ch, D, Q\} \xrightarrow{\theta} \{\text{nothing}, D, Q'\}$ <p>where <math>Q' = Q \cup \{ch \mapsto \emptyset\}</math></p> <p><b>SEND:</b></p> $\{\text{send } ch(x), D, Q\} \xrightarrow{\theta} \{\text{nothing}, D', Q\}$ <p>where <math>D' = D \cup \{\text{buff}(ch) = x\}</math></p> <p><b>IFTRUE:</b></p> $\frac{\langle expr \rangle = \text{true}}{\{\text{if } \langle expr \rangle \text{ then } q \text{ else } u, D, Q\} \xrightarrow{\theta} \{q, D', Q'\}}$ <p><b>LOOP:</b></p> $\frac{\{t, D, Q\} \xrightarrow{\theta'} \{t', D', Q'\}}{\{\text{loop } t \text{ end}, D, Q\} \xrightarrow{\theta} \{t'; \text{loop } t \text{ end}, D', Q'\}}$ <p>where <math>t</math> must contain a <code>sync</code> statement, or be rejected by the compiler</p> <p><b>SEQ1:</b></p> $\frac{\{t, D, Q\} \xrightarrow{\theta'} \{t', D, Q'\}}{\{t; u, D, Q\} \xrightarrow{\theta'} \{t'; u, D, Q'\}}$ <p><b>DPAR1:</b></p> $\frac{\{t, D, Q\} \xrightarrow{\theta'_i} \{t', D', Q'\} \quad \theta'_i - \theta_u \leq \delta_{u \rightarrow t}}{\{t, D, Q\} \langle \rangle \{u, D, Q\} \xrightarrow[\{\theta_i, \theta_u\}]{\{\theta'_i, \theta_u\}} \{t', D', Q'\} \langle \rangle \{u, D, Q\}}$ <p><b>CHECKABORT:</b></p> $\{\text{checkabort}(c, L), D, Q\} \xrightarrow{\theta} \{\text{nothing}, D, Q\}$ <p><b>ABORT1:</b></p> $\frac{\{t, D, Q\} \xrightarrow{\theta'} \{t', D', Q'\} \quad !c \mid \mid t \neq \text{checkabort}(c, L); t'}{\{\text{abort } t \text{ when } c:L, D, Q\} \xrightarrow{\theta} \{\text{abort } t' \text{ when } c:L, D', Q'\}}$ <p>where <math>:L</math> is the unique label of this abort statement</p>	<p><b>VAR:</b></p> $\{x = e, D, Q\} \xrightarrow{\theta} \{\text{nothing}, D', Q\}$ <p>where <math>D' = D \cup \{x = e\}</math></p> <p><b>EXPR:</b></p> $\{\langle expr \rangle, D, Q\} \xrightarrow{\theta} \{\text{nothing}, D, Q\}$ <p><b>READ:</b></p> $\langle ch \rangle(x), D, Q \xrightarrow{\theta} \{\langle value \rangle, D', Q\}$ <p>where <math>D' = D \cup \{\text{fresh}_{ch} = \text{false}\}</math> and <math>\langle value \rangle = D[\text{buff}(x)]</math></p> <p><b>SYNC:</b></p> $\{\text{sync } d, D, Q\} \xrightarrow[\theta]{\theta' = \theta + d} \{\text{nothing}, D', Q'\}$ <p>where for each <math>i \in \{1, \dots, d\}</math>:</p> <p><i>Freshen</i> : <math>D'_i = D'_{i-1} \cup \{\text{fresh}_q = \text{true} \mid q \in Q, \text{head}(q) \neq \perp\}</math></p> <p><i>Pop</i> : <math>D'_i = D'_i \cup \{q \mapsto \text{head}(q) \mid q \in Q\}</math></p> <p><i>Push</i> : <math>Q'_i = Q'_{i-1} \cup \{q \mapsto q \cup \text{buff}(q) \mid q \in Q\}</math></p> <p><b>IFFALSE:</b></p> $\frac{\langle expr \rangle = \text{false}}{\{\text{if } \langle expr \rangle \text{ then } q \text{ else } u, D, Q\} \xrightarrow{\theta} \{u, D', Q'\}}$ <p><b>FRESH:</b></p> $\{\text{fresh } ch, D, Q\} \xrightarrow{\theta} \{\langle expr \rangle, D, Q\}$ <p>where <math>\langle expr \rangle = D[\text{fresh}_{ch}]</math></p> <p><b>SEQ2:</b></p> $\frac{\{t, D, Q\} \xrightarrow{\theta} \{\text{nothing}, D, Q\}}{\{t; u, D, Q\} \xrightarrow{\theta} \{u, D, Q\}}$ <p><b>DPAR2:</b></p> $\frac{\{t, D, Q\} \xrightarrow{\theta'_i} \{\text{nothing}, D', Q'\}}{\{t, D, Q\} \langle \rangle \{u, D, Q\} \xrightarrow[\{\theta'_i, \theta_u\}]{\{\theta_i, \theta_u\}} \{u, D, Q\}}$ <p><b>ABORT2:</b></p> $\frac{\{t, D, Q\} \xrightarrow{\theta'} \{t', D', Q'\} \quad c \ \&\& \ t = \text{checkabort}(c, L); t'}{\{\text{abort } t \text{ when } c:L, D, Q\} \xrightarrow{\theta} \{\text{nothing}, D', Q'\}}$
---	--

contained term is aborted. Finally, Rule CHECKABORT captures the reduction of the `checkabort` statement itself to `nothing`. These rules are not explicitly used in this work, but form the basis for future work where they are used directly for compilation and verification.

#### 4 Determinism

Execution of a program consists of a series of *reactions*, which are atomic units of computation in Timetide, that delimit *instantaneous* computations done between progressions of the logical clock of a thread, demarcated by `sync` statements, unless the computation terminates.

Determinism has many interpretations, depending on the context and choices considered in the design of the system [12]. Determinism in Timetide is considered for each module of a program first. A Timetide module is deterministic when starting from the same initial state and inputs results in the same final state and outputs after any number of reactions.

In order to prove determinism of the overall program, we show that: ① each local module *reacts* to incoming tokens uniquely in any logical tick, and ② the buffers are confluent irrespective of the logical execution ordering of individual Timetide modules.

**Definition 2.** A reaction consists of a sequence of instantaneous statements, within any branch of a parallel thread, where the final transition follows a `sync d` statement, or is the term `nothing`. The reaction is denoted as:

$$\{t, D, Q\} \xrightarrow{\theta} \{t', D', Q'\} \xrightarrow{\theta} \cdots \xrightarrow{\theta} \{\text{sync } d; t''', D'', Q''\} \xrightarrow{\theta+d} \{t''', D''', Q'''\}$$

$$\text{or } \{t, D, Q\} \xrightarrow{\theta} \{t', D', Q'\} \xrightarrow{\theta} \cdots \xrightarrow{\theta} \{\text{nothing}, D'', Q''\}$$

In the following, reactions will be denoted using the following shorthand:

$$\{t, D, Q\} \xrightarrow{\theta'} \{t', D', Q'\} \text{ where } \theta' = \theta + d \text{ or } \theta' = \theta.$$

We first define the determinism of a reaction as follows.

**Definition 3.** Consider two reactions from the same initial state:

$$\{t, D, Q\} \xrightarrow[\{\theta\}]{\{\theta'\}} \{t'_1, D'_1, Q'_1\} \quad \text{and} \quad \{t, D, Q\} \xrightarrow[\{\theta\}]{\{\theta'\}} \{t'_2, D'_2, Q'_2\},$$

Where  $\{t, D, Q\}$  is the shared initial state, and  $\{t'_1, D'_1, Q'_1\}$  and  $\{t'_2, D'_2, Q'_2\}$  are the resulting states for two executions. Any such executions are deterministic iff the following is satisfied:

$$t'_1 = t'_2, \quad D'_1 = D'_2 \text{ and } \text{head}(q_1) = \text{head}(q_2) \quad \forall (q_1, q_2) \in (Q'_1, Q'_2)$$

The above requires that the residual, datastore, and head of the incoming channel queues ( $Q'_1$  and  $Q'_2$ ) is the same, in spite of the timing divergence.

We do not require the entire state of the inbound queues to be the same between two equivalent reactions, only their heads, as other values in the queue are unobservable to the receiving thread and hence do not affect the computation of the current reaction. Reactions progress in sequence to form an infinite *execution* of a Timetide program:

**Definition 4.** An execution of a parallel Timetide program starting at an initial state  $\{t, D, Q\} \langle \rangle \{u, D, Q\}$  is a sequence of reactions, denoted as:

$$\{t, D, Q\} \langle \rangle \{u, D, Q\} \xrightarrow[\{\theta\}]{\{\theta'_1, \theta''_1\}} \{t', D', Q'\} \langle \rangle \{u', D', Q'\} \xrightarrow[\{\theta\}]{\{\theta'_2, \theta''_2\}} \{t'', D'', Q''\} \langle \rangle \{u'', D'', Q''\} \cdots$$

Such a sequence is usually infinite, as the tasking model of Timetide uses infinite loops, but could also be finite in the case of preemption. Note that despite the syntax above, a prime does not necessarily indicate a change in either parallel term, as only one of the terms may have progressed in a single reaction (unless they both reduce at the same time).

Using this definition, we define determinism for an entire Timetide program as follows:

**Definition 5.** A Timetide **program** is deterministic iff for any two executions from the same initial state,

- (1) Every reaction has a corresponding reaction in the other execution with the same residual term, datastore, and head of the channel queues per Definition 3.
- (2) The reactions are partially ordered such that the order of each reaction associated with the same parallel thread is the same. However, the order of reactions between threads may differ.

Reactions within the same thread must be ordered, as programs are naturally sequential. However, the causality requirement between threads is relaxed up to a channel delay, as described by the side condition in Rule DPAR1. Consequently, it is acceptable for two executions to have different orderings of reactions between threads, so long as causal reactions are within the channel delay of each other. Despite the different orderings of reactions between threads, we prove that the behaviour of channel queues is confluent, meaning that the order in which values are pushed or popped from the queue does not affect the resulting computations.

**Lemma 1.** The behaviour of each individual reaction in a single thread is the same for each execution regardless of the ordering of reactions between different threads in a parallel program, because the queues are confluent.

**Proof.** Note that there are multiple rewrite choices for a parallel term:

- The left term can progress:  $\{t, D, Q\} \langle \rangle \{u, D, Q\} \xrightarrow[\theta]{\{\theta'_t, \theta_u\}} \{t', D', Q'\} \langle \rangle \{u, D, Q'\}$
- The right term can progress:  $\{t, D, Q\} \langle \rangle \{u, D, Q\} \xrightarrow[\theta]{\{\theta_t, \theta'_u\}} \{t, D, Q'\} \langle \rangle \{u', D', Q'\}$
- Both terms progress:  $\{t, D, Q\} \langle \rangle \{u, D, Q\} \xrightarrow[\theta]{\{\theta'_t, \theta'_u\}} \{t', D', Q'\} \langle \rangle \{u', D', Q'\}$

The value of the channel at the end of a logical tick may differ as a consequence of which arm is reduced. However, behaviour of any one reaction only depends on the values popped from the heads of the queues. Values may only be inserted in-order by appending to the tail of the queue (Rule SYNC). The only case where the tick of  $u$  could affect  $t$  is if the queue is empty, in which case the transition of  $t$  would not be enabled in the first place. Thus, when  $t$  eventually ticks, the value it pops from each  $head(q)$  is always the same regardless of the different possible transitions.  $\square$

Although the relative ordering of reactions is confluent, the individual reactions themselves must also be deterministic per Definition 3. We prove this by showing that each individual rewrite within a reaction is unique:

**Lemma 2.** A reaction in a Timetide program is finite and single-valued, meaning that it always reduces to a unique residual term

**Proof.** By structural induction on the term  $t$ .

- **Base case:** Suppose  $t = \text{nothing}$  or  $t = \text{sync } d; t'$ . In both cases, the reaction terminates immediately yielding  $\text{nothing}$  or the unchanged term  $t'$  respectively.

- **Case  $t = \text{loop } \tau' \text{ end}$ :** By Rule LOOP, all loops must contain at least one `sync` in every iteration. Therefore, any sequence of reductions from  $t$  will eventually encounter a `sync` statement, ending the reaction for the current tick.
- **Case  $t = u \langle \rangle v$ :** All possible interleavings of sub-terms are confluent per Lemma 1, and each sub-term  $u$  and  $v$  will eventually reduce to a unique residual term.
- **Other Cases:** Each base statement has a unique rewrite rule, except for branching statements `abort` and `if`, which may only vary as a result of input channel data which is confluent per Lemma 1. Hence, the residual is still uniquely determined.  $\square$

Thus, if each individual reaction is deterministic, and communications between reactions are confluent, then it follows that the overall global behaviour of the program is deterministic.

**Theorem 1.** *A program written with the Timetide semantics is deterministic.*

**Proof.** By contradiction. Assume that a non-deterministic program exists. This implies that in at least one reaction in the execution, a transition from a known state leads to two different subsequent states, i.e.:

$$\{t, D, Q\} \xrightarrow[\{\theta\}]{\{\theta'\}} \{t_1, D_1, Q_1\} \quad \text{and} \quad \{t, D, Q\} \xrightarrow[\{\theta\}]{\{\theta'\}} \{t_2, D_2, Q_2\}$$

Per Definition 3, this means that the residual terms  $t_1$  and  $t_2$  must be different, or the data store  $D_1$  and  $D_2$  must differ, or the heads of the channel queues must differ i.e.  $\forall (q_1, q_2) \in (Q_1, Q_2), \text{head}(q_1) = \text{head}(q_2)$

However, each reaction has a unique rewrite as shown in Lemma 2, which is also the only way the data store is updated. Furthermore, the heads of channel queues are always confluent per Lemma 1. Thus, the residual terms, data store, and heads of channel queues must be the same in both cases, contradicting the assumption that the program is non-deterministic.  $\square$

This definition of determinism has one major catch: the effects of physical time are disregarded. This is a recurring theme in the Logical Synchrony model, where we detach our synchronisation between threads from the physical time of the system. Interactions with the physical world are non-trivial and will be the subject of future work. However, we posit that there are still many applications where physical time is less critical than logical correctness, such as in distributed computation.

## 5 Synchronous Execution and Verification

Timetide models a purely synchronous abstraction of a distributed system. We have implemented a source-to-source compiler to convert Timetide into equivalent Esterel code. This supports two modes: a centralised mode which produces a single program for verification, and a distributed model which produces multiple communicating programs for deployment.

### 5.1 Distributed Target

In distributed mode, a separate Esterel module is produced for each thread of the Timetide program, which is subsequently compiled into a library that exposes a logical tick function. To execute these threads/modules  $\mathcal{T}$  over a distributed LSN  $\mathcal{L}$ , we need to define a mapping function  $\Gamma : \mathcal{T} \rightarrow V$  which says which LSN node (in  $V$ ) is used to execute each thread (in  $\mathcal{T}$ ). This mapping could be either manually defined or automatically generated (using techniques such as Integer Linear Programming (ILP) [14] or Simulated Annealing (SA) [3]) based on some constraints. For all communication channels, the delay through the LSN must be less than or equal to the Timetide delay, i.e.  $\forall_{\tau_1, \tau_2 \in \mathcal{T}} : \lambda_{\Gamma(\tau_1) \rightarrow \Gamma(\tau_2)} \leq \delta_{\tau_1 \rightarrow \tau_2}$ . In

the worst case, threads may need to be scheduled on the same node which incurs zero communication delay (but reduces parallelism). Additional constraints could be added based on system requirements, such as requiring tasks be executed on specific nodes due to available resources.

As a proof of concept in this work, we developed a lightweight wrapper that handles channel communication over network sockets. Each channel is modelled as a token-pushing FIFO buffer situated on the receiver end. To form the logical delays, each FIFO is pre-populated with a number of initial tokens. The main function simply performs a loop which pops from each inbound buffer, invokes the tick, and writes to each output socket. An abridged version of the generated wrapper is shown in Figure 11.

```

1  init_price_spread("tcp://localhost:1234"); // initialise channel
2  while(1) {
3      Signal price_spread = get_price_spread(); // Blocking read
4      if(price_spread.is_present()) { // Value or empty frame
5          in_price_spread(price_spread.val());
6      }
7      auto module_outputs = module_run(); // Run the esternel module
8      send_order(module_outputs.order); // Write to outbound channels
9  }

```

Fig. 11. Abridged generated C++ wrapper

**Lemma 3.** *The use of a FIFO with initial tokens faithfully implements a logical delay as defined in the Timetide semantics (Rules DPAR1 and DPAR2).*

**Proof.** By induction on the receiver clock  $\theta$ .  $\square$

## 5.2 Centralised Target

In the centralised compilation from Timetide, channels are synthesized as a number of intermediate signals, which form a shift register as shown in Figure 12

<pre> 1  chan &lt;id&gt; delay &lt;n&gt; in 2      run &lt;module&gt;[&lt;id&gt;/&lt;id&gt;]; 3      &lt;&gt; 4      run &lt;module&gt;[&lt;id&gt;/&lt;id&gt;]; 5  end chan </pre>	<pre> 1  signal &lt;id&gt;_0,...&lt;id&gt;_&lt;n&gt; in 2      run &lt;module&gt;[signal &lt;id&gt;_0]; 3         4      run &lt;module&gt;[signal &lt;id&gt;_n]; 5         6      loop 7          emit &lt;id&gt;_&lt;n&gt;(pre(&lt;id&gt;_&lt;n-1&gt;)); 8          pause; 9      end; 10        11     loop 12         emit &lt;id&gt;_&lt;n-1&gt;(pre(&lt;id&gt;_&lt;n-2&gt;)) 13     ... </pre>
--	--

(a) Timetide

(b) Esterel

Fig. 12. Translation of Timetide channels to Esterel signals

**Lemma 4.** *The use of cascaded `pre` operators in the centralised Esterel code faithfully implements a logical delay as defined in the Timetide semantics.*

**Proof.** By induction on the number of cascaded `pre` operators  $k$ .  $\square$

### 5.3 LSN-Compatible Architectures

The deterministic behaviour of a Timetide program is only relevant if it is executed on a suitable synchronisation layer. To this end, we define *LSN-compatible architectures* which allow for such deterministic execution. The minimum requirement for an LSN-compatible architecture is that the tick function from every thread in the Timetide specification may be called for an  $n$ th time only if the values from every inbound channel that were sent at logical time  $n-\delta$  are available, where  $\delta$  is the logical delay of that channel. It follows that because a Timetide program's behaviour solely varies upon the inputs it receives, then any architecture which guarantees the same input sequence will behave the same.

A number of approaches satisfy this requirement, and it is left up to the specific implementation to decide how it handles the buffers. For example, the bittide approach [25] is completely blocking-free (push-model), since it manages potential buffer overflow by self-balancing the clock speed of each distributed node. In comparison, the Kahn-like Finite FIFO Platforms (FFPs) [21] manages buffer overflow by using both blocking reads and writes (sometimes acting as a pull-model) across channels. Such blocking-based implementations can allow for an LSN to be implemented over generic networks such as TCP/IP.

### 5.4 Relation to the Physical World

Timetide programs live in the world of logical synchrony, meaning that the only reference to time is that of each node's local (synchronous) clock. While this is an advantage in the distribution, determinism, and verification of these programs, it typically restricts them to *closed* systems with no interface to the physical world. For some use-cases this is acceptable, such as closed-loop simulations, however for others this restriction needs to be relaxed.

There is one important question to keep in mind while contemplating the physical world: how does one guarantee determinism when each node has its own different view of the physical world, even at the same logic time? Allowing each individual node to interact with the physical world around it would lead to non-determinism, as signals which occur at the same physical time may be seen at different logical times, or vice versa. While it may be possible to bound this discrepancy using the LSN communication delays and the Worse Case Reaction Time (WCRT) of the application, this still poses problems.

To address this, we allow only a single node to interact with the environment, for physical time, inputs, or outputs. As a result, there is only one "view" of the physical world for the Timetide program, removing any potential paradoxical cases. For any other node which wishes to interact with the outside world, signals must travel from the node with the outside view, via paths of channels with defined logical delays in order to maintain determinism.

### 5.5 Model Checking of Timetide Programs

Because Timetide programs are semantically synchronous, despite the distributed execution, they may be easily verified using existing approaches as if they were a single program, due their equivalence as shown in Theorem 2. We do not consider the possibility of implementation-specific interference such as resource contention or pre-emption, as these are fundamentally implementation details to be solved separately of the Timetide model itself.

**Theorem 2.** *The execution of the distributed target for a Timetide program is equivalent to that of the centralised target.*

**Proof.** By induction on the logical clock  $\theta$  of a thread, and given Lemmas 3 and 4:

- At  $\theta = 0$  the initial states of both implementations are identical, and initial inputs match. The task code is identical in both implementations, thus outputs are identical.

- Assume that for  $\theta = k$  the internal state of both implementations are identical. At  $\theta = k + 1$ , the output is computed solely from the current state and the inputs of each inbound channel. Given that both implementations are equivalent, the inputs to the tasks are identical, hence so are the outputs.  $\square$

**5.5.1 Modelling.** Historically, safety properties of synchronous programs are verified using *synchronous observers*, which are programs that are composed in parallel with the main program that raise an alarm upon property violation. As opposed to an equally expressive Linear Temporal Logic (LTL) property, the condition of a synchronous observer is written in the same language as the program, making them simple to write and understand.

Consider again the trading system from Section 2. A safety property for this system may be that if a trader submits an order, it will not be missed by the centre. Intuitively, we might think this could be a problem since the trader runs more frequently than the exchange. However, this is addressed through the **fresh** operator. This property is encoded using synchronous observer shown in Figure 13. One way of encoding this is to ensure that the trade ID of the last order received by the exchange from a trader is exactly one greater than the previous order, otherwise it is implied that an order was missed. We compose the observer module in distributed parallel with the main program, as shown in Figure 14. Subsequently, in a model checker we simply verify that `PROPERTY_VIOLATED` is never emitted.

```

1 module observer:
2   input ack : integer;
3   output PROPERTY_VIOLATED : boolean;
4   var last_order := -1 : integer in
5     task(period=1, duration=1, offset=0):
6       if (fresh(ack)) {
7         if (ack != last_order+1) {
8           send PROPERTY_VIOLATED(true);
9         }
10        last_order := ack;
11      }
12    end task;
13  end var;
14 end module

```

Fig. 13. Safety Property Observer

```

1 ...
2 channel ack : integer delay 1;
3 channel PROPERTY_VIOLATED : boolean delay 0;
4 run Center(t1_order, t1_trade, t2_order, t2_trade, t1_spread, t2_spread, ack);
5 <>
6 run Trader(t1_spread/price_spread, t1_order/order, t1_trade/trade, 0/id);
7 <>
8 run Trader(t2_spread/price_spread, t2_order/order, t2_trade/trade, 1/id);
9 <>
10 run observer(ack, PROPERTY_VIOLATED);

```

Fig. 14. A synchronous observer composed with the main program

These properties which pertain to the timing of events are much easier to verify than more general safety properties. Because the LET tasks execute periodically, they necessarily form a repeated hyperperiod of

execution [17], following some initial prelude. Thus, it is sufficient to verify the program over a single (non-prelude) hyperperiod, which captures all possible interactions between the tasks.

**5.5.2 Verification Results.** For demonstration, we have implemented several synchronous observers for applications including the trading system as well as a model of basic cruise controller, shown in Figure 15a. For this work we simply use the bounded model checker CBMC [23] on the actual generated code, which is sufficient for safety and bounded-liveness properties. An example of the bounded liveness property is shown in Figure 15b.

<pre> 1  chan speed : float delay 1, 2  tht1      : float delay 2, 3  rpm       : float delay 1, 4  setpoint  : float delay 1, 5  in 6  run ShaftSensor(speed, rpm); 7  &lt;&gt; 8  run Ctrl(speed, tht1, setpoint); 9  &lt;&gt; 10 run Motor(tht1, rpm); 11 end chan; 15 %padding </pre>	<pre> 1  module LivenessObserver: 2  input setpoint, speed : float; 3  output LIVE_VIOLATED : boolean = false; 4  var counter := 0 : integer in 5  task(period=1,duration=1): 6  if (fresh(setpoint)) { counter := 0; } 7  if (setpoint != current_speed) { 8  counter = counter + 1; 9  } else { counter = 0; } 10 if (counter &gt;= 10) { 11 send LIVE_VIOLATED(true); 12 } 13 end task 14 end var 15 end module </pre>
---	---

(a) The toplevel code of the cruise control

(b) The Timetide code of the bounded liveness property

Fig. 15. The cruise control example and a synchronous observer

Results are shown in Table 4, including a property which fails to validate the verification process. Because this implementation is a simple proportional controller, there is no explicit prevention of negative speed, which was correctly identified by the model checker.

Table 4. Verification Results

Program	Property Description	Result
Trading	No missed orders	PASS
Trading	Stock cannot be overtraded	PASS
Cruise Controller	Not more than 5% over target speed	PASS
Cruise Controller	Target speed reached within 10 ticks	PASS
Cruise Controller	Speed can never be negative	FAIL

## 6 Results

As the closest related language, we compare the performance and correctness of Timetide with that of Lingua Franca. The example applications were implemented in both languages, using the same host calls, and the throughput and trace equivalence of the two languages were compared. All benchmarks were run locally on a 2021 Macbook Pro with a 10-Core M1 Pro chip. Network latency was simulated using the tool *Comcast* [36]. To create an equivalent Lingua Franca system, we have the following translation rules: Each distributed thread becomes a reactor with the same inputs, outputs, and variables. A ‘fresh’ boolean

variable is added for each input. Each task becomes a periodic reaction, with an additional reaction to freshen each input. Each logical tick is translated into 1 ms of physical time. The important characteristic is the relative task lengths, so the unit of time is not important. A minimal example of the translation from Timetide to Lingua Franca is shown in Figure 16.

<pre> 1 module example: 2   input a : integer; 3   output b : integer; 4   var x : integer = 0 in 5     task(duration = 2, period = 3): 6       if fresh(a){ 7         x = a; 8         send b(x); 9       } 10    end task; 11  end var; 12 end module 15 %padding </pre>	<pre> 1 reactor example{ 2   input a : int; 3   output b : int; 4   state x : int = 0; 5   state fresh_a : bool = false; 6   timer t(0,3ms); 7   reaction(t) a -&gt; b {= 8     if (fresh_a) { 9       self-&gt;x = a; 10      fresh_a = false; 11      lf_set(b, self-&gt;x) 12    } 13  =}; 14  reaction(a) {=self-&gt;fresh_a=true;=}; 15 } </pre>
(a) Timetide	(b) Lingua Franca

Fig. 16. Translation of a Timetide module to Lingua Franca

Furthermore, instantiations in Timetide are directly translated to reactor instantiations in a Lingua Franca, as shown in Figure 17. The task duration is added to the transmission delay in the Lingua Franca program, as reactors take zero logical time to execute.

<pre> 1 channel ch1 : integer delay 3; 2 run example[ch1/b] &lt;&gt; run example[ch1/a]; 3 %padding </pre>	<pre> 1 example e1 = new example(); 2 example e2 = new example(); 3 e1.b -&gt; e2.a after 5ms; % dur. + delay </pre>
(a) Timetide	(b) Lingua Franca

Fig. 17. Translation of module instantiation from Timetide to Lingua Franca

Using our financial trading example, we ensure that the resulting output trace between the two languages is equivalent. The external function calls are exactly the same, so the only difference is the coordination infrastructure provided by the language. When applying the same seed to the random number generators in each language and logging the outputs, we confirm that the observable traces of the two implementations are identical.

Lingua Franca has two distributed coordination schemes: the default, which coordinates distributed threads with a central arbiter, and a decentralised peer-to-peer coordinator. To avoid confusion with the non-distributed variant, we will refer to these two flavours as *Arbiter LF* and *P2P LF*. Both of these are compared to the Timetide approach. The Timetide approach is more similar to the P2P LF approach. However, P2P LF cannot run unrestricted by physical time, unlike Arbiter mode, as the P2P distribution relies on physical timing to synchronise. Consequently, to give a fair benchmark we iteratively increased the physical-time gearing (slowing down the program) of the P2P LF nodes until the Lingua Franca program stopped reporting timing errors, in order to get tight bounds on the fastest execution rate. We ran the financial trading system for 10,000 logical ticks using both the Timetide and Lingua Franca

generated code and measured the total execution time. The results are shown in Figure 18, averaged over 3 executions per delay variation.

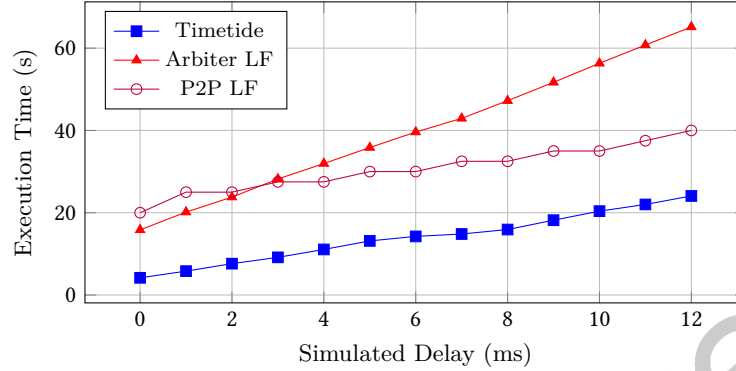


Fig. 18. Throughput comparison between Timetide and Lingua Franca

Although Lingua Franca is undoubtedly the more mature compiler and execution time appears to be roughly linear for both tools, the throughput of the Timetide generated code is significantly (around 3×) faster than the Arbitrator variant and almost twice as fast as the P2P variant. This is likely due to two major factors:

- (1) Timetide compiles to baremetal C and communicates over a lightweight wrapper. In comparison, Lingua Franca relies on a runtime which relates logical and physical time. Despite disabling physical time restrictions, the overhead of the runtime remains.
- (2) Timetide automatically pipelines its execution when transmission delays are specified, which the arbitrator flavour of Lingua Franca doesn't do.

To examine Timetide's capability in building large-scale systems, we have implemented a model of a sensor network. This sensor network consists of *leaf nodes* which sense data from the environment, *aggregator nodes* which combine data from local leaf nodes into a single data point, and a *central node* which ultimately receives data from all aggregator nodes. An example with two aggregator nodes, each with three leaf nodes, is shown in Figure 19. Compared with the trading example, this example demonstrates hierarchy in Timetide. Moreover, the use of sensor nodes with a low latency to a local compute node, and a larger latency link to a central node, is a common pattern in modern edge computing. Here, we match the periods of all nodes to the same value, so that the production and consumption of sensor data is matched. An abridged version of the Timetide code is shown in Figure 20.

To demonstrate how Timetide handles programs with a larger number of tasks, we measure the execution time while varying the number of aggregator nodes (**AGGS**), where each additional aggregator node adds four tasks to the system. The results are shown in Figure 21, using a latency of 500 ms between each aggregator node and the main hub.

For one and two aggregator nodes we have 5 and 9 threads respectively, meaning that each thread can be assigned to a core on the CPU. Consequently, we see no performance penalty, even with the additional communication. For three or more aggregator nodes, the number of threads exceeds the number of cores and consequently we see a relatively linear increase in execution time, showing that system performance does not degrade exponentially with additional threads. Lingua Franca will spawn local worker threads for instantiated reactors within other reactors, but not in a distributed fashion, and seems to scale poorly

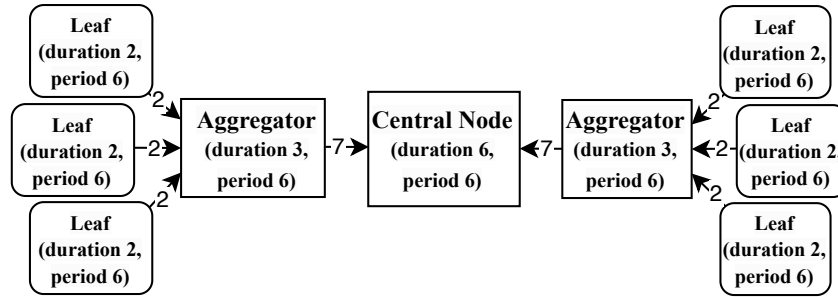


Fig. 19. Block diagram of a sensor network with two aggregator nodes which each have three leaf nodes

<pre> 1 module toplevel: 2   const AGGS : int = 2; 3   channel from_agg : int[AGGS] delay 7; 4 5   run CentralNode[from_agg] 6   &lt;&gt; 7   foreach i in AGGS { 8     run Aggregator[i, from_agg[i]] 9   } 10 end module; 15 %padding                 </pre>	<pre> 1 module Aggregator: 2   const LEAVES : int = 3; 3   output from_agg : int; 4   channel from_leaf : int[LEAVES] delay 2; 5 6   foreach i in LEAVES { 7     run LeafNode[i, from_leaf[i]] 8   } &lt;&gt; 9   task(duration=3, period=6): 10    var aggregate : float = aggregate_readings( 11      ↪ from_leaf) in 12    send from_agg(aggregate); 13  end task; 14 end module;                 </pre>
--	---

(a) Top level module

(b) Aggregator Node

Fig. 20. The sensor network example in Timetide

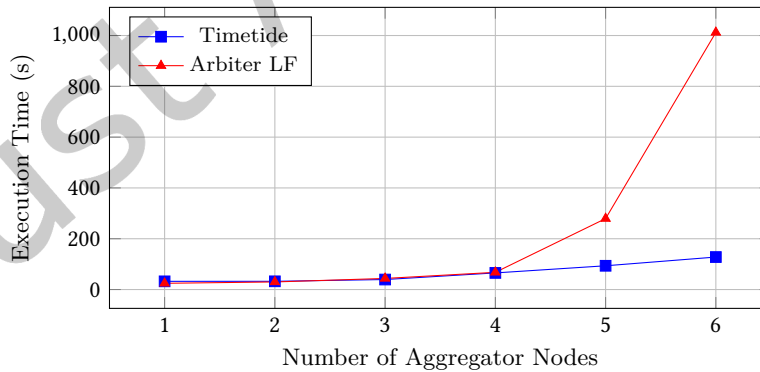


Fig. 21. Sensor Network Throughput on a 10-Core M1 Pro Macbook with Varying Aggregator Nodes

with the number of communicating threads. For comparison, running the same 6-arbiter program on the *centralized* Lingua Franca runtime yielded an execution time of just 66.6 s.

We also compare the two toolchains by their compilation speed, number of lines of code (albeit, according to our own translation rules), and the size of the generated binaries. Binary size is the sum of all produced binaries for a distributed program, however the separately-installed Lingua Franca Runtime Interface is not included. For these comparisons, we use our financial exchange and sensor network examples along with two (periodic) examples from the Lingua Franca playground. The results are shown in Table 5.

Table 5. Comparison of Compilation Speed, Number of Lines of Code, and Binary Size

	Exchange		CAL		Rosace		Sensor Network	
	TT	LF	TT	LF	TT	LF	TT	LF
Num. Lines	~110	~130	~50	~50	~400	~400	~100	~100
Num. Threads	3	3	4	4	11	4	9	3
Sum Bin. Size (kB)	1938	337	2045	396	6250	425	4780	298
Compile Time (s)	5.4	22.3	6.0	22.8	26.4	26.8	17.1	21.5

The compilation speed of Timetide is significantly faster than Lingua Franca for these examples. However, the Lingua Franca compilation time does not tend to vary significantly for different programs — much of the time is taken compiling the appreciable runtime. In comparison, the syntax translation from Timetide to Esterel takes negligible time and consequent Esterel compilation is similarly rapid. The number of lines of code is not meaningfully different, as the constructs of each language have similarities. The size of the generated binaries is somewhat larger in Timetide than in Lingua Franca, likely because the code generated by the dated Esterel v5 compiler is not particularly compact. Interestingly, the Timetide implementations of the Rosace and Sensor Networks benchmarks produce many more distributed threads, due to the nesting of instantiations actually spawning additional threads. In comparison, only top-level instantiations are distributed in a Lingua Franca system. In any case, no concerning behaviour was observed in the above metrics for either language; the primary difference is the logical versus physical model of synchronisation.

Ultimately, we are comparing the performance of Timetide within its niche to Lingua Franca operating outside its intended domain of real-time systems. However, we are not aware of any other languages which provide deterministic execution across a distributed system to compare against. Many of the Lingua Franca playground examples can simply not be expressed in Timetide due to the lack of support for input from the environment. Hence, the Timetide model is less expressive than Lingua Franca, in exchange for its simplified synchronisation model. Thus, we conclude that the Timetide model of logical synchrony is the most performant choice for systems which require determinism and high throughput, but do not necessarily have real-time or reactivity requirements.

## 7 Related Work

The synchronous paradigm was independently introduced through three foundational languages: Esterel [7], Lustre [18] and Signal [6]. These rely on the *synchrony hypothesis*, assuming a reactive system operates infinitely fast relative to its environment. A logical tick function is repeatedly invoked, which samples inputs from the environment and computes the outputs. Synchronous programs typically *compile away concurrency* to produce sequential code which scales well, even for large systems. However, distribution of synchronous programs is challenging [4, 16], due to the expense of distributing a global clock. Many works instead aim to focus on making synchronous programs *latency-insensitive* [9] to avoid the need for

synchronisation, however this is limited to a small subset of synchronous programs. The *N-synchronous model* [10, 31] relaxes synchrony by allowing threads to be desynchronised by at most  $n$ -ticks using a FIFO buffer. In  $N$ -synchrony, buffer sizes need to be computed based on a schedule and synthesized instead being specified, unlike Timetide. Our approach relies on implicit buffers, which are left to the communication protocol of the underlying implementation, such as bittide [25] or Finite FIFO Platforms (FFP) [37]. Table 6 summarises some languages that enable deterministic distribution, based on how they model time, synchronise, specify task duration, and express communication delay.

Table 6. Deterministic languages for distributed systems

Language	Model of Time	Synchronisation	Task duration	Latency
Multiclock-Esterel [8]	logical	HW sync.	multiples of ticks	implicit
Giotto [19]	physical	clock sync.	zero	instantaneous
PsyC [35]	logical	clock sync.	multiples of ticks	instantaneous
Lingua Franca [30]	logical+physical	clock sync.	zero	optional fixed delay
Timetide	logical	logical sync.	multiples of ticks	fixed delay

The languages Giotto and PsyC are both used for the modelling of LET systems, but assume insignificant communication delays for distribution, which is not realistic but is instead abstracted away during scheduling on a physical device. The exception to this is the recent federated flavour of Lingua Franca [29], which specifies transmission delays but either requires a centralised coordinator or leverages strong guarantees on the physical clock synchronisation of the network, neither of which are required by Timetide.

## 8 Conclusions

A deterministic programming model for distributed systems remains elusive, except for the recent federated flavour of Lingua Franca [29]. However, even this model relies on physical clock synchronisation, which suffers from scalability issues. To address this gap, we introduce the Timetide language for distributed systems, which treats communication delay as a first-class citizen based on the logical synchrony approach. This allows the programmer to reason about the system as if it were centralised and synchronous, where the transmission delays of a system are expressed in a fixed number of logical ticks, rather than physical time.

We introduce and formalise the semantics of Timetide and show how it can be used to model distributed systems, also demonstrating a structural translation to the synchronous language Esterel. In doing so, we show that the distributed Timetide model can be verified using conventional approaches. Furthermore, we describe a class of *Logical Synchrony Network Compatible Architectures* which can implement Timetide programs in a distributed setting.

While this work provides the first logically synchronous programming model, our work has some limitations: There is currently no process to automatically map Timetide programs to distributed LSNs, and a concrete way to relate the logical world of the Timetide model to physical time and environmental inputs is not provided. Additionally, there can be a need for tasks to have variable execution rates or even be enabled/disabled entirely, which Timetide currently does not support. We will dwell on these limitations in the near future.

## Acknowledgments

The authors from the University of Auckland acknowledge the support of the Google grants *Designing Scalable Synchronous Applications over Google bittide* and *Towards a formally verified autonomous vehicle by leveraging the bittide protocol*. They also acknowledge the biweekly meetings with the Google bittide team, namely Tammo Spalink, Sanjay Lall, and Martin Izzard.

## References

- [1] [n. d.]. Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. IEEE standard 2021.9456762.
- [2] Sidharta Andalam, Partha Roop, Alain Girault, and Claus Traulsen. 2009. *PRET-C: A new language for programming precision timed architectures*. Ph.D. Dissertation. INRIA.
- [3] Gamal Attiya and Yskandar Hamam. 2006. Task allocation for maximizing reliability of distributed systems: A simulated annealing approach. *Journal of parallel and Distributed Computing* 66, 10 (2006), 1259–1266.
- [4] Daniel Baudisch, Jens Brandt, and Klaus Schneider. 2010. Dependency-driven distribution of synchronous programs. In *IFIP Working Conference on Distributed and Parallel Embedded Systems*. Springer, 169–180.
- [5] 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 (2003), 64–83.
- [6] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. 1991. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of computer programming* 16, 2 (1991), 103–149.
- [7] Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming* 19, 2 (1992), 87–152.
- [8] Gérard Berry and Ellen Sentovich. 2001. Multiclock esterel. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 110–125.
- [9] Luca P Carloni, Kenneth L McMillan, and Alberto L Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *IEEE Transactions on computer-aided design of integrated circuits and systems* 20, 9 (2001), 1059–1076.
- [10] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. 2006. N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. *ACM SIGPLAN Notices* 41, 1 (2006), 180–193.
- [11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google’s globally distributed database. *ACM Transactions on Computer Systems* 31, 3 (Aug. 2013), 1–22. doi:10.1145/2491245
- [12] Stephen A Edwards. 2018. On determinism. *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday* (2018), 240–253.
- [13] Rolf Ernst, Leonie Ahrendts, and Kai-Björn Gemlau. 2018. System level LET: Mastering cause-effect chains in distributed systems. In *IECON 2018-44th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 4084–4089.
- [14] Christodoulos A Floudas and Xiaoxia Lin. 2005. Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications. *Annals of Operations Research* 139 (2005), 131–162.
- [15] KAHN Gilles. 1974. The semantics of a simple language for parallel programming. *Information processing* 74, 471–475 (1974), 15–28.
- [16] Alain Girault. [n. d.]. A survey of automatic distribution method for synchronous programs.
- [17] Mario Günzel, Kuan-Hsun Chen, Niklas Ueter, Georg von der Brüggen, Marco Dürr, and Jian-Jia Chen. 2021. Timing analysis of asynchronized distributed cause-effect chains. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 40–52.
- [18] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.
- [19] Thomas A Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. 2001. Giotto: A time-triggered language for embedded programming. In *Embedded Software: First International Workshop, EMSOFT 2001 Tahoe City, CA, USA, October 8–10, 2001 Proceedings 1*. Springer, 166–184.
- [20] Charles Antony Richard Hoare et al. 1985. *Communicating sequential processes*. Vol. 178. Prentice-hall Englewood Cliffs.

- [21] Logan Kenwright, Partha Roop, Nathan Allen, Sanjay Lall, Călin Caşcaval, Tammo Spalink, and Martin Izzard. 2024. Logical Synchrony Networks: A formal model for deterministic distribution. *IEEE Access* (2024).
- [22] Christoph M Kirsch and Ana Sokolova. 2012. The logical execution time paradigm. *Advances in Real-Time Systems* (2012), 103–120.
- [23] Daniel Kroening and Michael Tautschnig. 2014. CBMC–C Bounded Model Checker: (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5–13, 2014. Proceedings 20*. Springer, 389–391.
- [24] Sanjay Lall, Călin Caşcaval, Martin Izzard, and Tammo Spalink. 2024. Logical Synchrony and the bittide Mechanism. *IEEE Transactions on Parallel and Distributed Systems* (2024), 1–14. doi:10.1109/TPDS.2024.3444739
- [25] Sanjay Lall, Călin Caşcaval, Martin Izzard, and Tammo Spalink. 2022. Modeling and Control of bittide Synchronization. In *2022 American Control Conference (ACC)*. IEEE, 5185–5192.
- [26] Edward A Lee. 2021. Determinism. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 5 (2021), 1–34.
- [27] Edward A Lee, Ravi Akella, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. 2023. Consistency vs. availability in distributed cyber-physical systems. *ACM Transactions on Embedded Computing Systems* 22, 5s (2023), 1–24.
- [28] Yuliang Li et al. 2020. Sundial: fault-tolerant clock synchronization for datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1171–1186.
- [29] Marten Lohstroh, Soroush Bateni, Christian Menard, Alexander Schulz-Rosengarten, Jeronimo Castrillon, and Edward A Lee. 2024. Deterministic coordination across multiple timelines. *ACM Transactions on Embedded Computing Systems* 23, 5 (2024), 1–29.
- [30] 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 (2021), 1–27.
- [31] Louis Mandel, Florence Plateau, and Marc Pouzet. 2010. Lucy-n: a n-synchronous extension of Lustre. In *Mathematics of Program Construction: 10th International Conference, MPC 2010, Québec City, Canada, June 21–23, 2010. Proceedings 10*. Springer, 288–309.
- [32] Robin Milner. 1984. Lectures on a calculus for communicating systems. In *International Conference on Concurrency*. Springer, 197–220.
- [33] Gordon D Plotkin. 2004. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming* 60 (2004), 3–15.
- [34] Fabien Siron, Dumitru Potop-Butucaru, Robert De Simone, Damien Chabrol, and Amira Methni. 2022. The synchronous Logical Execution Time paradigm. In *ERTS 2022-Embedded real time systems*.
- [35] Fabien Siron, Dumitru Potop-Butucaru, Robert De Simone, Damien Chabrol, and Amira Methni. 2023. *Formal Semantics of the PsyC language*. Ph. D. Dissertation. Inria-Sophia Antipolis.
- [36] Tyler Treat. 2025. comcast. <https://github.com/tylertreat/comcast> original-date: 2014-11-12.
- [37] Stavros Tripakis, Claudio Pinello, Albert Benveniste, Alberto Sangiovanni-Vincent, Paul Caspi, and Marco Di Natale. 2008. Implementing synchronous models on loosely time triggered architectures. *IEEE Trans. Comput.* 57, 10 (2008), 1300–1314.

## A Artifact Appendix

### A.1 Abstract

A docker image containing the transpiler which converts a Timetide program to a (distributed) Esterel program, as well as the source code for the corresponding case studies. This is an earlier version of the compiler that was used for benchmarking, so there are minor syntax differences to the paper. The docker image contains an Ubuntu 20.04 setup with the environment variables correctly set for the compiler to run.

### A.2 Artifact check-list (meta-information)

- **Binary:** `tt2str1` Source to Source Compilation, included in docker image
- **Run-time environment:** Dockerized Ubuntu 20.04
- **Output:** Esterel code, C code, Makefiles, Binary files
- **How much disk space required (approximately)?:** 2GB
- **How much time is needed to prepare workflow (approximately)?:** 2 minutes
- **How much time is needed to complete experiments (approximately)?:** 5 minutes
- **Publicly available?:** Binary - Yes, Source - Future work
- **Code licenses (if publicly available)?:** MIT
- **Workflow framework used?:** Docker
- **Archived (provide DOI)?:** 10.5281/zenodo.16251935

### A.3 Description

A.3.1 *How to access.* Available from Zenodo at <https://doi.org/10.5281/zenodo.16251935>

A.3.2 *Software dependencies.* Docker capable of running Ubuntu 20.04.

### A.4 Installation

- Ensure Docker is installed and running on your device
- Run `docker load -i tt_compiler_linux.tar` to install the image to Docker. Then, to spawn a fresh instance of the CLI, use `docker run --rm -it --platform linux/amd64 tt_compiler_linux`.

### A.5 Experiment workflow

Upon loading the docker image you will find yourself in a CLI prompt inside the `"/compiler/"` folder. This folder contains:

- The `tt2str1` binary, built for x86\_64 Linux.
- Bash scripts to build and run all three of the main case studies.
- A bundled copy of the academic Esterel v5 compiler, used internally in the Timetide compiler.
- The examples folder, each example containing a `.tt`` file which contains the Timetide code

The three examples may be built by running the bash script `./build_all_examples.sh`. Because this builds the distributed flavour with many binaries, this may take some time (1-2 minutes).

Following the build completion, you may use any of `./run_cruise_controller.sh`, `./run_exchange.sh`, or `./run_sensors.sh` to run the case studies. These are set up to run for 100,000 logical ticks on each node, and then print the time elapsed node.

### A.6 Evaluation and expected results

The expected results are the time elapsed on each node, printed to the console. By default we aren't exposing any particular variables or outputs, but the user may modify the `main.cpp` of each module after compilation (and subsequently re-run `make`) to print out any variables they wish to observe.

Received 10 August 2025; revised 10 August 2025; accepted 19 August 2025

Just Accepted