2024 22nd ACM-IEEE International Symposium on Formal Methods and Models for System Design (MEMOCODE)

Logical Synchrony Plus Functional Processes Entail Observable Determinacy

Sanjiva Prasad
IIT Delhi, India
sanjiva@cse.iitd.ac.in

***Abstract*—Determinacy is a desirable but difficult-to-achieve behavioural property in scalable distributed systems. Deterministic Models of Computation range from the asynchronous Kahn Process Networks to synchronous reactive languages such as Lustre, where logical clocks enforce the synchrony hypothesis. These models have well-founded data-flow semantics where computations are viewed as the least fixed point solutions of simultaneous equations defined by continuous functions on streams of discrete values. However, scalable and efficient implementations of the Kahn model are challenging to construct, while the synchrony hypothesis in Lustre makes distributed implementations difficult. Moreover, determinacy is a consequence of specific assumptions built into the computational model.**

The notion of *Logical Synchrony*, proposed by Lall *et al.*, and explored further by Kenwright *et al.*, suggests that synchronisation issues may be decoupled from computation, leading to a distributed model where computations at independent nodes are related by invariant logical delays. We provide a semantic notion of behaviour for functional processes running on such Logical Synchrony Networks (extension graphs), and an appropriate and robust notion of logical observational equivalence (wavefront equivalence) retaining semantic aspects of KPNs, specifically determinacy.

Further, we propose extending the versatile notion of the synchronous observer, exploited in the Lustre toolset, to a network of located synchronous observers with the same invariant logical delays as the distributed system. Thus we will be able to use the same logically synchronous model of computation for checking or monitoring a class of (safety) properties of programs, specifying axioms and assumptions on behaviour, constraining models and specifying test cases, etc.

***Index Terms*—: Data Flow, Determinacy, Kahn Process Networks, Logical Synchrony, Lustre, Synchronous Observer, Synchronous Product**

I. Introduction

Models of computation, which specify how compositions of concurrent (possibly distributed) computational processes communicate with one another, vary greatly in their approach to *synchrony*. These "MoCs" range from lock-step systolic computation and synchronous communication to completely asynchronous processes that communication with indeterminate delays. (See e.g., [1] for a discussion on MoCs.) *Determinacy* is an important property about system behaviour since it implies predictability and repeatability. However, achieving scalable and determinate behaviour of complex distributed systems is challenging.

Asynchronous models are commonly associated with *indeterminacy* in behaviour, due to the vagaries of scheduling and network delays. A notable exception is the eponymous Kahn Process Networks model (KPNs) [2], [3], where a network of asynchronously executing nodes performing functional computations over values that are sent (written) and received (read) between the processes over unidirectional FIFO channels exhibit determinate observable behaviour. Determinacy is obtained due to specific constraints on the data streams and continuity of the functional operations. Importantly, computation at a node *blocks* when performing a receive operation until a value appears on that input channel, whereas processes are not constrained about sending on channels. "Blocking receives" achieve a degree of synchronisation between the asynchronously executing nodes (and consequently determinacy) in KPNs, but at a significant performance cost, adversely impacting the efficiency of implementations. (KPNs are briefly reviewed in section II.)

In contrast, the paradigm of synchronous reactive systems [4], exemplified by Esterel [5], [6], [7], Lustre [8], [9], and Signal [10], [11], achieves determinacy by subjecting the execution of the system to a (set of) logical clocks, and enforcing the *synchrony hypothesis*. However, distributing a common clock is not scalable. A major benefit of the synchronous paradigm is that it supports a rich suite of formal methods and tools for checking or monitoring a class of (safety) properties of systems, specifying axioms and assumptions on their behaviour, constraining model behaviour, and specifying test cases, etc. These leverage the versatile notion of *Synchronous Observer*, discussed in section III, that is expressible *within* the same programming framework [12], [13].

Wouldn't it be wonderful if magically inputs would always be present in buffers (so there would be no blocking on receives) and yet there were no need for explicit global temporal synchronisation? Lall *et al.* posit that the *bittide* system with its distributed "reframing protocol" achieves such "syntony" between sender and receiver processes by using a dynamic control system where each node makes only local observations and maintains only small buffer offsets with little state information [14]. Underlying bittide is the interesting notion of *Logical Synchrony* Networks proposed by Lall *et al.* [15], and explored further by Kenwright *et al.* [16], which suggests that synchronisation issues may be decoupled from computation, leading to a distributed model where computations at independent nodes are related by *invariant logical delays* (discussed in section IV).

In this paper, we explore a model of computation where a distributed functional data-flow system executes on such a *logical synchrony network* (LSN). We present the behaviour extensionally as a labelled graph that is observably deterministic, and where causal dependencies are explicitly captured (see section V). We sketch how such "Kahn Logically Synchronous Networks" (KLSNs) may be simulated in the synchronous framework of Lustre (cf. [17]). Next, we show that our semantics is consistent with the notion of equivalence of LSNs defined in [15] by proposing a precise relationship, which we call "wavefront equivalence", between the extension graphs of two equivalent LSNs, preserving observable behaviour upto the relative adjustment of clock indices at the different nodes, and also addressing the attendant issue of initialisation of data-flow streams.

These ideas suggest that the notion of *logical synchrony* may provide the appropriate abstraction that spans these two *deterministic* data-flow MoCs – the asynchronous KPNs [2], [3] and the synchronous Lustre model [8], [9] – where functional computations are expressed declaratively as a set of (possibly recursive) data flow equations. The significance of both these MoCs is that they exhibit the important property, dubbed the "*Kahn Principle*", that the *compositional semantics* of a network of communicating processes can be characterised as the least fixed point of continuous functionals over channel histories [18]. The ability to simulate a KLSN in a synchronous reactive framework (Lustre) implies determinacy, as well as a method for developing and reasoning about KLSNs using the Lustre tool suite. Further, it suggests that we may be able to develop a theory of a distributed logically synchronous observation of behaviour by adapting the notion of *synchronous observers*: one could place synchronous observers at each node of a KLSN and build a composite logically synchronous observer for certain classes of properties.

II. Kahn Process Networks

In his seminal 1974 paper, Gilles Kahn studied the semantics of a simple computational model consisting of a network of asynchronously executing processes performing functional computations over values that are sent (written) and received (read) between the processes over unidirectional FIFO channels. Such eponymous Kahn Process Networks (KPNs) can thus be abstractly represented as directed graphs, the nodes of which correspond to processes, and where the edges correspond to directional FIFO channels between the processes. Kahn observed: "*Arbitrary interconnection of systems, as well as processes, is legitimate. Hence, top-down design finds here a mathematical justification since we can postpone the decision to implement a given function by a single process or a set of interconnected processes: this decision will not introduce perturbations in the remainder of the system.*"

The main assumptions on the model are that (i) the communication channels between processes are the only way for the processes to synchronise; (ii) that the communication may involve a finite but indeterminate delay; (iii) at any instant, a process is either computing *or* waiting for input on exactly one of its incident input channels; (iv) the processes are sequential, i.e., deterministic.

Kahn posited that such networks have very clean mathematical semantics over domains that are complete partial orders on finite or denumerably infinite sequences of values: The network behaviour is the least fixed point solution to a system of simultaneous equations

$$X_i' = f_i(X_1, \ldots, X_n)$$

which relate output channel histories with input channel histories via a set of Scott-continuous functions

$$f_i : [D_1^\omega \times \ldots \times D_n^\omega \to D_i'^\omega],$$

thus allowing the use of Scott's Induction Principle for reasoning. A subsequent paper [3] showed that these data-flow semantics are consistent with operational semantics of co-routine execution of the processes, as well as demand-driven lazy execution of a functional program. Continuity of the functions precludes a process requiring an infinite input history in order to begin producing its output, while monotonicity implies that more input cannot result in less output. Reasoning about KPNs relies on these facts: (i) input buffers of processes are not bounded and outputs do not block, (ii) any output produced by a process depends only on *previously* received inputs, not on any input received later than or simultaneously with that output, and (iii) consistent input histories produce consistent outputs.

The remarkable character of KPNs is that despite asynchrony between the processes and arbitrary delays in communication, their behaviour is *deterministic*. Apart from obvious constraints from the assumptions — namely that at each individual node the computation is a function that operates instantaneously on the values from all its incident input (channels) and that every channel has exactly one producer node, that the channels obey a FIFO discipline, etc. — determinacy is achieved by insisting that computation at a node *blocks* when performing a receive (read operation) on an input channel until a value appears on it (a process cannot be simultaneously waiting at one or another input channel). Relaxing these constraints leads to complications such as the Brock-Ackerman anomaly [19], where the mathematical semantics is inconsistent with intuitive operational behaviour.

III. Synchronous Reactive Data-Flow

In the synchronous reactive systems paradigm [4], a system is in continual interaction with its environment, reacting instantly and deterministically to environment events; moreover, its output is often intended to influence the environment. Lustre [8], [9] is a synchronous data-flow language used for modelling, simulating, and verifying a variety of systems including embedded controllers, safety-critical systems, communication protocols, railway signal networks, avionics, etc.

A Lustre program consists of a set of module ("node") definitions, whose inputs and outputs are *clocked* data streams, where a flow takes its n^{th} value on the n^{th} clock tick. A node definition consists of a set of equations, of the form $\vec{x} = \vec{e}$

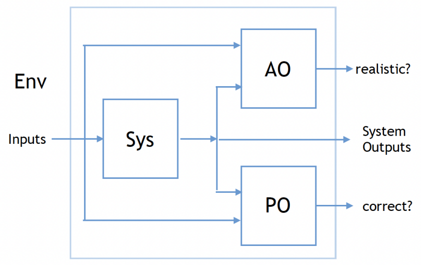


Fig. 1. Synchronous Observation of a System

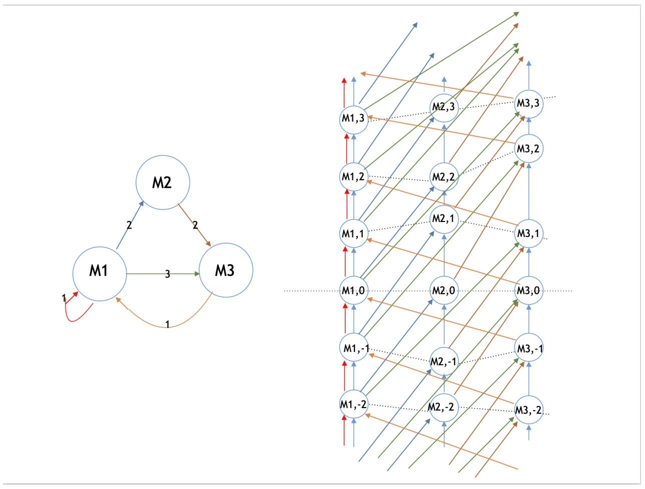


Fig. 2. A Logical Synchrony Network and its Extended Graph

which *define* flows associated with output and local variables, and are intended to express *temporal invariants* between input and output flows. Expressions e specify flows, and include constants, variables, expressions using pointwise unary and binary operators, conditionals, and node calls (module instantiations). In addition, there are temporal operators; here we focus only on two (both deprecated) operators: `pre` and `->`, and do not consider clock-dependent and sampling operations. While streams may be recursively defined (provided they can be "well-clocked"), module definitions are not recursive. As in the case of KPNs, the semantics of Lustre programs is formally defined (co-inductively) in terms of data-flow streams domains [20], with the various constraints – particularly those related to clocking – enforcing determinacy.

Not only does Lustre's model exhibit the Kahn Principle, it supports the notion of a *synchronous observer*, namely a module adjoined to a system model that continuously monitors system (and environment) state variables and produces a boolean stream which indicates whether a specified condition is satisfied or violated. Synchronous observers, especially for safety properties, can be written within the *same* framework as the system. Model-checking for synchronous languages is efficient since synchronous product is exactly parallel composition. Apart from monitoring a system for safety properties (as stated above), synchronous observers can be used to check assumptions about or enforce constraints on the environment in which a system operates. The synchronous observer helps focus the model-checking effort to only those cases which are interesting. Since observers are executable, they are can be used to specify and generate test cases.

Figure 1 shows a system Sys coupled with a property observer PO that observes the input streams from the environment Env as well as the output streams from the system, pronouncing whether the system behaves correctly or not. In addition, we have a synchronous observer AO that checks assumptions on the inputs from the environment as well as system outputs (which may influence further environment inputs), judging whether these are relevant or realistic. The combination of Sys, PO and AO support a methodology for checking a variety of safety properties.

IV. Logical Synchrony

Lall *et al.* recently proposed an event model called *logical synchrony* [15], which is sufficient to capture notions of causality in a network of processes, but without the need for a global clock or a universal notion of time. Each node executes events at its *own pace* without explicitly synchronising with other processes, maintaining its own *local clock* (not necessarily equitempered in its ticks, and in fact decoupled from physical time) to sequentially index events at that location. Operationally, at each local logical tick, node i inserts a communication frame at the end of each of its outgoing FIFO edges, while also consuming a frame from the front of each of its incoming FIFO edges.

Clock progressions at different locations are not comparable. However, events at different nodes are related by the causal order imposed by communication. This notion of logical synchrony differs slightly from Lamport's notion of logical time and clocks [21], in that while the same causal ordering relation of events is captured, there is no need for time-stamping and adjustment of logical clocks, since the objective isn't to create a global notion of logical time consistent with the causal order. The authors also claim that this notion adequately characterises the causal relationships underlying the synchronisation patterns in other paradigms, including those in the polyglot Lingua Franca [22].

A *Logical Synchrony Network* (LSN) can be abstractly characterised as a *directed multigraph* $G = (N, E, \lambda)$, the nodes N representing computing machines and the directed edges E between nodes representing FIFO communication channels between them. Associated with each edge $e \in E$ is an integer $\lambda(e)$, representing an *invariant logical delay* between its source and target nodes ($\lambda : E \to \mathbb{Z}$, so this delay may even be negative). The idea is that if e is an edge from node i to j, then an event logged at node i's m^{th} tick of its logical clock results in sending data to node j (via the communication channel represented by directed edge e) which will be received at node j in *its* n^{th} logical clock tick, where $n = m + \lambda(e)$.

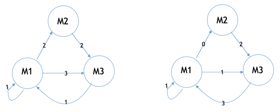


Fig. 3. Equivalent LSNs with offsets $c_1 = 0, c_2 = -2, c_3 = -2$

And further, the $(m+1)^{th}$ event at i will causally affect the $(n+1)^{th}$ event at j, and so on.

Logical delays are additive along a directed path:

$$\lambda(P) = \Sigma_{e \in P} \lambda(e)$$

This sum is path dependent, and so two paths with the same endpoints may have different logical delays.

We have relaxed some of the constraints in [15, Defs 1,2]: (i) we allow multiple edges between two nodes, reflecting the possibility of multiple channels between two nodes, with possibly different logical delays; (ii) we allow self-loops on nodes but only with positive labels, reflecting the forward implicit "communication" of the state of local variables from one instant to a future instant.

Based on this directed graph, one can construct an infinite "extended (multi)graph" $G_e = (N_e, E_e, \lambda)$ capturing causality of events: its set of nodes $N_e = \{(i, m) \mid i \in N, m \in \mathbb{Z}\}$. The edges E_e are of two kinds: (1) *Communication edges*: $\{(i, m) \to (j, m + \lambda(e)) \mid e = i \overset{\lambda(e)}{\to} j \in E\}$, i.e., for each $e \in E$ from i to j, there is a directed edge from each node (i, m) to (j, n), where $n = m + \lambda(e)$; (2) *Computation edges*: $\{(i, m) \to (i, m+1) \mid i \in N\}$.[1] Figure 2 shows a simple LSN and its extended graph with colour-coded edges; the lack of alignment and the variation in edge lengths is deliberate.

If the (cumulative) "round-trip delays" around *each* directed cycle C in a LSN G are positive i.e, $\lambda(C) > 0$, then the extended graph G_e is acyclic. A negative round-trip delay would imply a violation of causal ordering of events. The causal ordering of events is the *transitive closure* of the edges in such an acyclic G_e.

An intriguing concept in [15] is the notion of equivalent LSNs, based on relabelling of edges: two LSNs L_1, L_2 on the same directed multigraph $G = (N, E)$ but with different edge labellings λ and λ' are called (LSN-)equivalent, written $L_1 \sim L_2$, if there exist constants $c_1, \ldots, c_{|N|} \in \mathbb{Z}$ such that for each $e \in E$ from node i to j, $\lambda'(e) = \lambda(e) + c_j - c_i$.[2] Figure 3 shows two equivalent LSNs.

In the extended graphs, the (communication) edges $(i, m) \to (j, n)$ in the first LSN are mapped to $(i, m + c_i) \to (j, n + c_j)$ in the second LSN. While this idea seems at first blush to scramble timelines, it *preserves causal relations between events*, because it is only a translational shift in the index numbering of events wrt the local logical clocks at the nodes. Note that this *translation* of edge labels depends only on the *clock translations* at the end-points, and therefore, for *any* path P from any node n_1 to any n_2, $\lambda'(P) = \lambda(P) + c_{n_2} - c_{n_1}$. Thus (i) for any directed cycle C, the round-trip delay is unchanged, i.e., $\lambda'(C) = \lambda(C)$, so this notion of equivalence preserves acyclicity of the *extended (multi)graph* G_e, despite the relabelling of clock ticks. From a practical viewpoint, we can use relabelling to *entirely avoid negative edge labels*. Further, for two different paths P_1 and P_2 from any node n_1 to any n_2, $\lambda'(P_1) - \lambda'(P_2) = \lambda(P_1) - \lambda(P_2)$, i.e., the *relative differences* in path delays are preserved[3].

V. Kahn Logical Synchrony Networks

We now explore a MoC consisting of running functional data-flow computations given by a system of equations $\{x_i = f_i(\vec{x})\}$ on LSNs, an idea suggested but not explored further in [15], since the focus there was on developing the idea of logical synchrony and LSNs. We also address the issue of initialisation mentioned there, in kick-starting a data-flow functional computation on a LSN. We call such a model *Kahn Logical Synchrony Networks* (KLSNs), characterised by a multigraph $K = (N, E, \lambda, F)$, where F is a system of data-flow equations. We discuss here only the simple case, where associated with each node i is a single defining equation $x_i = f_i(\vec{x})$, and as in LSNs, the logical clocks at nodes progress in logical syntony, with edge labels being non-negative.[4]

Consider for instance a system involving the stream functions specified by the equations $x_1 = f_1(x_1, x_2, x_3) = x_1 + x_3$, $x_2 = f_2(x_1, x_2, x_3) = 2 * x_1$, and $x_3 = f_3(x_1, x_2, x_3) = x_1 * x_2$, where, since the stream for variable x_1 is recursively defined, the x_1 in the rhs needs to be a prior value of x_1, and x_1 therefore needs an initialisation, set here arbitrarily to the value 1. A synchronous rendering of this data-flow computation in Lustre is:

```
x1 = 1 -> (pre(x1)+ pre(x3))
x2 = 2 * x1
x3 = x1 * x2
```

Mapping this system onto a LSN, we get a labelled multigraph with (for simplicity in this treatment) each f_i mapped to a node i, and for each output stream x_i defined by f_i, an edge $e_{x_i:i \to j}$ from the node i producing x_i to each node j consuming x_i. Now since the LSN associates via λ each such edge $e_{x_i:i \to j}$ to a logical delay, we will observe on the output streams quite different behaviour from the preliminary Lustre encoding above. Displayed in Figure 4 is this data-flow program mapped onto the LSN in Figure 2, as well as its Lustre encoding. Note that in the encoding, we have altered the number of `pre` operators as well as indicated (in bold brown) initial values on the streams.[5] These initial values may be chosen arbitrarily (discussed later). Also presented at the right of Figure 4 is the observed behaviour of such a KLSN with the given

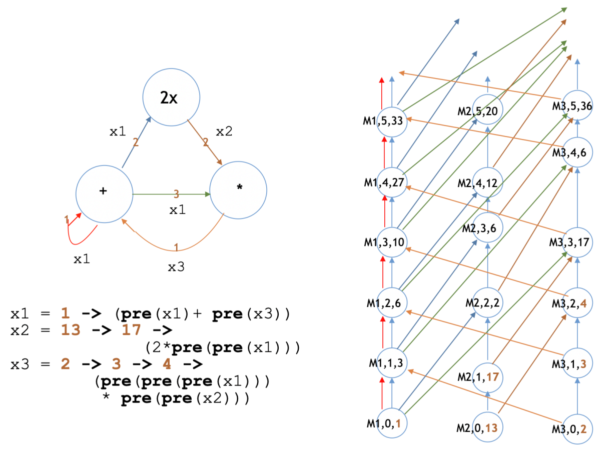
[1]If we place self loops on each node in $i \in N$, labelled with delay 1, computation edges are special cases of communication edges.

[2]Yes, Virginia, $\sim$ is an equivalence!

[3]This is only hinted at via an example in [15].

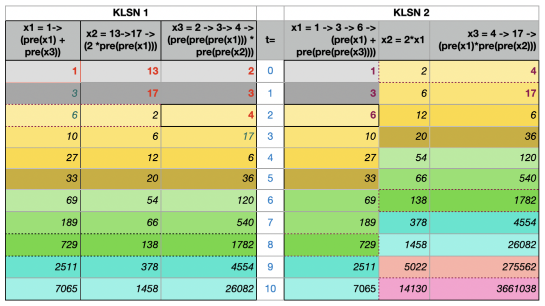
[4]As observed in the previous section, this is not a limitation.

[5]Here the initialisations of `x1` to `[1]`, `x2` to `[13, 17]`, and `x3` to `[2,3,4]` were arbitrarily chosen; with a different initialisation, the observed streams would be different.



```
x1 = 1 -> (pre(x1)+ pre(x3))
x2 = 13 -> 17 ->
              (2*pre(pre(x1)))
x3 = 2 -> 3 -> 4 ->
        (pre(pre(pre(x1)))
         * pre(pre(x2)))
```

Fig. 4. Example KLSN with Extension Graph and Lustre encoding



KLSN 1				KLSN 2		
x1 = 1-> (pre(x1) + pre(x3))	x2 = 13->17 -> (2 *pre(pre(x1)))	x3 = 2 -> 3-> 4 -> (pre(pre(pre(x1))) * pre(pre(x2)))	t=	x1 = 1 -> 3 -> 6 -> (pre(x1) + pre(pre(pre(x3))))	x2 = 2*x1	x3 = 4 -> 17 -> (pre(x1)*pre(pre(x2)))
1	**13**	**2**	0	**1**	2	**4**
3	**17**	**3**	1	**3**	6	**17**
6	2	**4**	2	**6**	12	6
10	6	17	3	10	20	36
27	12	6	4	27	54	120
33	20	36	5	33	66	540
69	54	120	6	69	138	1782
189	66	540	7	189	378	4554
729	138	1782	8	729	1458	26082
2511	378	4554	9	2511	5022	275562
7065	1458	26082	10	7065	14130	3661038

Fig. 5. Wavefront Equivalent KLSNs with offsets $c_1 = 0, c_2 = -2, c_3 = -2$

initial values. We call this an *extension graph*, to highlight that it represents the observable (extensional) behaviour of the system, and also to allude to its basis on the "extended graph" of a LSN. Let $X[m]$ denote the value at index m of the stream history X for variable x. The nodes of the *extension graph* $EGKL$ are $(i, m, X_i[m]) \mid i \in N, m \in \mathbb{N}\}$ where X_i is the *channel histories* for x_i.[6] The edges, from $(i, m, X_i[m])$ to $(j, n, X_j[n])$, correspond exactly to the underlying LSN delay as specified by λ for the edge $e_{x_i:i\to j}$ between nodes i, j , and the functional dependencies between values $X_i[m]$ and $X_j[n]$ in the channel histories, as specified by the function f_j in the system of equations F.

We can simulate the KLSN in a Lustre-like language by performing the following transformation: (i) If in the KLSN, $\lambda(e_{x_i:i\to j}) = d$, (where $d \geq 0$), then in the equation $x_j = f_j(\vec{x})$, each occurrence of x_i is preceded by d `pre` operators. (ii) Moreover, if in $x_j = f_j(\vec{x})$, k is the maximum nesting depth of `pre` operators, then the expression for $f_j(\vec{x})$ will be preceded by an initialisation sequence v_1 `->` ... `->` v_k`->` of k initial values of the appropriate type.

Mapping a well-formed set of data-flow equations onto a LSN with positive cycles (i.e, an acyclic extended graph) gives us the following result:

Proposition 1: The observable behaviour of the synchronous encoding of a KLSN corresponds exactly to its extension graph (modulo the initialisations). □
Consequences of this proposition are that since Lustre programs have deterministic semantics, KLSNs are determinate in their behaviour; moreover, the Lustre tool suite becomes available for reasoning about KLSN systems.

Now what happens if we were to map the *same* functional data-flow equations to an equivalent LSN, related by the clock offsets $c_1, \ldots, c_j$? Naïvely observed, the outputs of the corresponding KLSNs would be different at the local logical clock ticks. However, there is a rather pleasant relationship between the observed behaviours:

[6]Note that the indices m, n are natural numbers, and so not negative.

Proposition 2: If LSNs $L_1 \sim L_2$ on the same multi-graph $G = (N, E)$, related via logical clock translations $(c_1, \ldots c_n)$, are both identically decorated with functional data-flow equations from $F = \{x_i = f_i(\vec{x})\}$, then there exist some initialisations $\vec{v}_i$ (for each $i \in N$) such that in the corresponding extension graphs $EGKL_1$ and $EGKL_2$, $(i, m, X_i[m]) \in EGKL_1 = (i, m{+}c_i, X_i[m{+}c_i]) \in EGKL_2$ (provided $m, m + c_i \in \mathbb{N}$; else the equation is irrelevant). □
We call this induced notion of correspondence between extension graphs wrt distributed observations a "wavefront equivalence". Figure 5 illustrates the wavefront equivalence between the same example functional data-flow program on the two equivalent LSNs of Figure 3 with offsets $0, -2, -2$. The induced Lustre encodings are shown in the column headings for comparison. The "wavefronts" have been colour-coded, so the reader can see correspondence between the relevant cells. Certain values, such as $x_2[0] = 13, x_2[1] = 17$ and $x_3[0] = 2, x_3[1] = 3$ in the first KLSN have no corresponding entries in the second KLSN, whereas $x_2[9], x_2[10]$, $x_3[9], x_3[10]$ in the second KLSN correspond to values that will appear later in the first. The bold-face entries are chosen initialisation values – either arbitrarily, or in the case of $x_1[1] = 3, x_1[2] = 6$ in the second KLSN picked to match the corresponding entries in the first.

VI. Discussion

Logical Synchrony Networks present an insightful notion that allows us to abstract from the notion of physical time, as well as clockwork synchronisation between different computational nodes. We have sketched a model we call KLSN for performing Kahn-like functional data-flow computations on a LSN, with extension graph semantics and a notion of "wavefront equivalence" that is consistent with the notion of LSN equivalence. Our ideas provides some credence, at least for two well-studied models of deterministic data-flow computation, to the claim in [15] that logical synchrony allows distributed computing to be coordinated as tightly as in synchronous systems but without the distribution of a global clock or reference to universal time.

We have indicated how KLSNs may be encoded in a Lustre-like framework. The immediate consequence of this encoding is determinacy and also that we may readily adapt a variety of formal methods and tools from the synchronous programming world. Kenwright *et al.* have been developing tools for KLSNs, inspired by the Lustre suite of tools.

In particular, the notion of synchronous observer discussed in section III may find a natural extension from clocked synchrony to logical synchrony. We conjecture that since synchronous observers can be rendered as data-flow programs on streams, they too can be mapped onto a given LSN: This allows us to build a distributed logical observer network, with local logical synchronous observers at each node indicating whether a local property has been preserved/violated and constraining the local environment at that node. The local observers may then be logically synchronously coordinated quite efficiently to check global behavioural properties of the system. Interesting questions include whether the logical delays between observers should be in sync with those of the system being observed, and what class of safety properties may be monitored in this manner.

Weaker notions of observational equivalence can be developed for notions of LSN and KLSN equivalence – for instance on non-identically shaped LSNs, such as homeomorphic variations, and where observations on some of the nodes are hidden or ignored. We hope that our preliminary ideas for the semantics and the coordinated yoking of local logically synchronous observers to a KLSN will give us constructive methods for composition of KLSNs. Specifically we would like to explore sufficient but general conditions under which compositions of KLSNs behave in an orderly fashion, e.g., preserving determinacy, and whether a principled "Assume-Guarantee" methodology can be articulated.

The deeper semantic roots of our proposal rely (of course) on the Kahn Principle, and requires further study.

Acknowledgments

The ideas presented here originated during the author's visit in March 2024 to Auckland University, supported by a fellowship from the IIT Delhi - New Zealand Universities Centre, during which he had fruitful discussions with Logan Kenwright, Partha Roop and Nathan Allen. He is grateful to them for their generosity, for introducing him to the work of Lall *et al.*, and their own work in developing LSNs, and for including him in their conversations with Sanjay Lall, Călin Casçaval and Tammo Spalink.

References

[1] S. A. Edwards, L. Lavagno, E. A. Lee, and A. L. Sangiovanni-Vincentelli, "Design of embedded systems: formal models, validation, and synthesis," *Proc. IEEE*, vol. 85, no. 3, pp. 366–390, 1997.

[2] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974* (J. L. Rosenfeld, ed.), pp. 471–475, North-Holland, 1974.

[3] G. Kahn and D. B. MacQueen, "Coroutines and Networks of Parallel Processes," in *Information Processing, Proceedings of the 7th IFIP Congress 1977, Toronto, Canada, August 8-12, 1977* (B. Gilchrist, ed.), pp. 993–998, North-Holland, 1977.

[4] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proc. IEEE*, vol. 79, no. 9, pp. 1270–1282, 1991.

[5] G. Berry, "The foundations of Esterel," in *Proof, Language, and Interaction, Essays in Honour of Robin Milner* (G. D. Plotkin, C. Stirling, and M. Tofte, eds.), pp. 425–454, The MIT Press, 2000.

[6] G. Berry and L. Cosserat, "The ESTEREL Synchronous Programming Language and its Mathematical Semantics," in *Seminar on Concurrency, Carnegie-Mellon University, Pittsburg, PA, USA, July 9-11, 1984* (S. D. Brookes, A. W. Roscoe, and G. Winskel, eds.), vol. 197 of *Lecture Notes in Computer Science*, pp. 389–448, Springer, 1984.

[7] G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, 1992.

[8] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: A Declarative Language for Programming Synchronous Systems," in *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pp. 178–188, ACM Press, 1987.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, vol. 79, pp. 1305–1320, Sep. 1991.

[10] A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous Programming with Events and Relations: the SIGNAL Language and Its Semantics," *Sci. Comput. Program.*, vol. 16, no. 2, pp. 103–149, 1991.

[11] T. Gautier and P. Le Guernic, "SIGNAL: A declarative language for synchronous programming of real-time systems," in *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings* (G. Kahn, ed.), vol. 274 of *Lecture Notes in Computer Science*, pp. 257–277, Springer, 1987.

[12] N. Halbwachs, F. Lagnier, and P. Raymond, "Synchronous Observers and the Verification of Reactive Systems," in *Algebraic Methodology and Software Technology (AMAST '93), Proceedings of the Third International Conference on Methodology and Software Technology, University of Twente, Enschede, The Netherlands, 21-25 June, 1993* (M. Nivat, C. Rattray, T. Rus, and G. Scollo, eds.), Workshops in Computing, pp. 83–96, Springer, 1993.

[13] J. M. Rushby, "The Versatile Synchronous Observer," in *Formal Methods: Foundations and Applications - 15th Brazilian Symposium, SBMF 2012, Natal, Brazil, September 23-28, 2012. Proceedings*, vol. 7498 of *Lecture Notes in Computer Science*, p. 1, Springer, 2012.

[14] S. Lall, C. Cascaval, M. Izzard, and T. Spalink, "Modeling and Control of bittide Synchronization," in *American Control Conference, ACC 2022, Atlanta, GA, USA, June 8-10, 2022*, pp. 5185–5192, IEEE, 2022.

[15] S. Lall, C. Cascaval, M. Izzard, and T. Spalink, "Logical Synchrony and the bittide Mechanism," *CoRR*, vol. abs/2308.00144, 2023.

[16] L. Kenwright, P. S. Roop, N. Allen, S. Lall, C. Cascaval, T. Spalink, and M. Izzard, "Logical Synchrony Networks: A Formal Model for Deterministic Distribution," *IEEE Access*, vol. 12, pp. 80872–80883, 2024.

[17] N. Halbwachs and L. Mandel, "Simulation and Verification of Asynchronous Systems by means of a Synchronous Model," in *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pp. 3–14, 2006.

[18] E. W. Stark, "Concurrent Transition System Semantics of Process Networks," in *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pp. 199–210, ACM Press, 1987.

[19] J. D. Brock and W. B. Ackerman, "Scenarios: A Model of Non-Determinate Computation," in *Formalization of Programming Concepts, International Colloquium, Peniscola, Spain, April 19-25, 1981, Proceedings* (J. Díaz and I. Ramos, eds.), vol. 107 of *Lecture Notes in Computer Science*, pp. 252–259, Springer, 1981.

[20] T. Bourke, L. Brun, and M. Pouzet, "Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset," *Proc. ACM Program. Lang.*, vol. 4, Dec. 2019.

[21] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[22] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee, "Toward a Lingua Franca for Deterministic Concurrent Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 4, pp. 36:1–36:27, 2021.