TICKTOCKTOKENS: a minimal building block for event-driven systems

Johannes Leugering
Bioengineering Dept.
UC San Diego, USA

***Abstract*—Event-driven computing is a key feature of many neuromorphic circuits, yet we lack a common general purpose language to describe such systems, reason about their capabilities, and design them by deliberately composing smaller parts. In this paper, I want to ask: what is the equivalent of a logic gate for event-driven systems? I propose one very simple candidate and show how it can be composed into arbitrarily complex event-driven systems using a framework I call TICKTOCKTOKENS. I motivate its design, discuss its capabilities, and share a hardware implementation as a very compact digital circuit.**

***Index Terms*—event-driven computing, neuromorphic, discrete event system, tokens, building block**

I. INTRODUCTION

Event-driven computing is one of the core tenets of neuromorphic engineering, and widely seen as a key ingredient for achieving brain-like energy efficiency in electronic circuits. But what do we really mean with "event-driven computing"? Unlike 'conventional' computer science, which is firmly rooted in (clocked) Boolean logic and finite state machines (FSMs), and unlike physics, which relies heavily on differential equations, we have not yet converged on a similar common language to construct and model event-driven systems. In other words: what is the NAND-gate of event-driven computers?

In this paper, I want to show what I think could be a simple atomic building block for constructing event-driven systems – an event-driven gate, so to say, that can be composed into various event-driven logic circuits. This concept, which I named TICKTOCKTOKENS (TTT) for reasons that may become apparent in the following, can model event-driven systems and lends itself to an efficient implementation in digital hardware. I share Verilog code of a VLSI implementation that I developed as part of the TinyTapeout project [1] in 130 nm CMOS Skywater technology.

My objectives for this paper are three-fold:

1) to motivate a simple, constructive and compositional formalism for event-driven computing based around the TTT-concept.
2) to show how this formalism can be applied to model various event-driven systems, including but not limited to spiking neural networks,
3) and to demonstrate how such a component can be realized as a highly integrated digital circuit with "neuromorphic qualities".

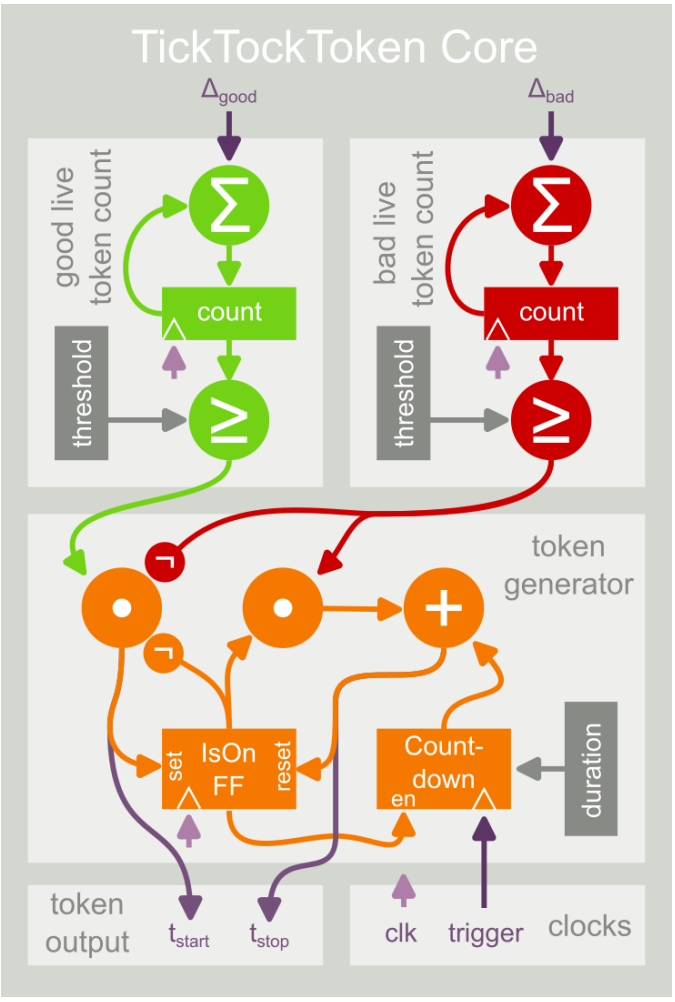


Fig. 1: Schematic overview of a TickTockToken core. There are two counters for good and bad incoming tokens, which are compared against programmable thresholds to decide whether the node should start or stop emitting its token. Once triggered, the token stays active for a finite duration or until it is stopped by bad token input. If the token lifetime is very long, the countdown can be triggered by a much slower clock.

A. Motivation

This project started with a practical question: how can we formalize dendritic computation? We had developed a simple model of dendritic plateau potentials and how they enable single neurons to detect complex temporal patterns [2]. Now we wanted to formalize its behavior [3], analyze its capabilities [4], and finally derive efficient hardware implementations, but we struggled to find an established framework for modeling such a system. Most biophysical neuron models use systems of continuous (stochastic, partial) differential equations, but in our case, the dendritic compartments only require *discrete* states, and the interaction is purely *event-driven*, i.e. it occurs at discrete yet irregularly spaced points in time.

The standard frameworks for modeling discrete state systems are FSMs and Petri Nets [5], but they lack an explicit notion of time, i.e. they cannot express that one event can (or must) happen within some specific time-interval after another - an essential feature to understand e.g. coincidence detection. There are variants of these frameworks, namely Timed Automata/FSMs [6] and Time(d) Petri Nets [7], that introduce a notion of time[1]. And indeed, it is possible to analyze networks of (multi-compartment) spiking neurons in these formalisms [3], [8]–[10].

But since they are primarily used for *verification* of complex systems, they lack other features that are essential for *implementing* event-driven computing systems, in particular *composability* and *parallelism*. Both FSMs and Petri Nets treat the entire system, no matter how large, as a single state (vector), which evolves one transition at a time. This poses a challenge if we want to build up a distributed system from smaller sub-systems, each of which should run (and change state) simultaneously.[2]

Given the limitations of these common frameworks, I believe we need a better language and better tools to describe computation in event-driven systems. In particular, I'd like such a framework to satisfy the following requirements:

1) It should model a system composed of *nodes* that communicate exclusively by (repeatedly) triggering a finite set of events, and otherwise operate independently of each other.
2) In order to perform any non-trivial computation[3], each node must maintain some form of internal memory or *state*. The entire system's state is the union of all its nodes' states and nothing else.
3) The nodes' internal memory should be finite or fading [11]. Here, I'll further assume it to be discrete-valued.
4) Each node should be time-invariant, i.e. delaying all incoming events delays the node's state and outputs accordingly.
5) Computation is *local*, i.e. a node's behavior is fully determined by its own inputs and internal state (which, in turn, is determined by its previous inputs). Conversely, this implies that each node can independently update its internal state or trigger events regardless of other nodes' internal states.
6) Finally, the system should be able to perform "non-trivial" computations; for example, it should be able to detect and generate temporal sequences of events.

[1]They are continuous-time models in the sense that timing constraints can be real-valued (or at least rational), but discrete-time in the sense that there is a finite set of events in any finite amount of time.

[2]In the context of *verification* this is not an issue, because the model can be given as much time as necessary to process all events in (arbitrary) order. But for a *generative* model this is a problem, because without global arbitration two parts of the system could simultaneously make local state transitions that are mutually exclusive.

[3]Without memory, the node can only process perfectly synchronous events, which occur with zero probability in any real system subject to noise and timing variations.

B. Related work

I am certainly not the first to model event-driven systems, so I've drawn inspiration from a lot of prior work. First and foremost, the neuromorphic and theoretical neuroscience community at large has developed many different approaches to implement event-driven systems like SNNs and event-based sensors [12]. Specifically under the umbrella of *spike-time coding* [13], various frameworks have been developed to model how information can be represented by the (relative) timing of spikes [14], [15], how spiking neurons such as leaky integrate-and-fire neurons can compute with spikes [11], and even how this could be learned through plasticity mechanisms [16]. But most of these approaches focus on explaining the specific computational capabilities of SNNs, whereas I'm interested in modeling event-driven systems more abstractly, including the inner mechanism of individual (multi-compartment) neurons.

Another school of thought, going back at least to Von Neumann [17], McCulloch and Pitts [18] and Minsky [19] has been to analyze the capabilities of (discrete) neural networks from the perspective of formal logic [20] and, more generally, function approximation (e.g. [21]). These approaches, however, don't yet address the asynchronous, real-time characteristics of *event-driven* systems such as spiking neural networks.

Outside the neuromorphic community, event-driven or discrete event systems [22] are also studied for process modeling and the *verification* of parallel and distributed systems [5]. These domains mainly make use of variants of Timed FSMs and Time(d) Petri Nets [7], which suffer from the aforementioned limitation that in order to guarantee correctness, they model state transitions as occurring one-at-a-time (with arbitrarily high temporal resolution). This serial execution means that although these frameworks can be used to *analyze* the behavior of spiking neural networks [3], [8]–[10], they are less suitable for actually implementing them.

Other frameworks from theoretical computer science include communicating sequential processes [23], communicating FSMs [24] and communicating hardware processes [25], all of which are designed to model event-driven computer systems, but they make use of specific concepts such as message queues and sequential programs which don't translate well to non-von Neumann systems.

A more suitable approach for constructing neuromorphic systems might be race-logic and Temporal State Machines

[26], an algebraic framework that can be used to formulate algorithms at a high-level of abstraction and translate them into an event-timing-based VLSI implementation. Here, however, I'd like to explore a more bottom-up approach, and instead start with a minimal hardware building block.

The only other framework I am aware of that follows a similar approach towards building event-driven systems is the STICK [27] framework and its extension STEAM [28]. These models also construct event-driven micro-circuits from simple parts, but they use heterogeneous neuron-like building blocks with continuous dynamics. Here, I will instead consider a discrete system.

II. The TickTockTokens model

A. Idea

The very simple idea underlying the TickTockTokens concept is that each node only has a binary state, which it can broadcast to other nodes in an event-driven way by signaling each rising and falling transition with a specific event t^i_{start} or t^i_{stop}, respectively. In analogy to the Time Petri Net framework, I think of a node's `start` event as lending a *token* to each connected neighbor, who then stores it inside a *place*. This token is later recalled by the corresponding `stop` event.

Each connection to a neighbor node is endowed with a signed weight. Depending on the sign, the tokens lands in one of two places: for negative weights, tokens are accumulated in the *bad tokens* place; for positive weights, in the *good tokens* place. The (integer) magnitude of the weight dictates, how many "copies" of each token to put into the respective place - equivalently, we could allow multiple connection between the same nodes.

A node then turns `on` once it holds enough good tokens and not too many bad tokens. Once triggered, the node emits its `start` event and remains in the `on` state until it either "expires" after some finite amount of time, or it is stopped by too many bad tokens; it then emits its `stop` event and reverts back to the `off` state. The expiration is handled by an internal countdown timer, which provides the framework with a notion of time.

The tokens of different nodes are indistinguishable, so to keep track of the received tokens, each node only needs to increment (decrement) a token counter for every incoming start (stop) event by the magnitude of the weight. Each node comprises two such counters, one for good tokens and one for bad tokens. How many good or bad tokens are required to cause the node's state to flip is determined by a respective good or bad token *threshold* - two programmable parameters of the node.

If we want to implement this in a clocked circuit, we can integrate the net flow of good and bad tokens for the entire clock cycle into two integer values, Δ_{good} and Δ_{bad}. Fig. 1 shows a schematic of how one TickTockToken node can be implemented.

By connecting many such nodes and setting the thresholds and weights appropriately, we can implement arbitrarily complex "event-driven logic" circuits! In the following, I will derive this more formally and then show a few simple examples.

B. Derivation

Each node i has a binary state variable $x_i \in \{\texttt{off}, \texttt{on}\}$, which is initialized to an `off` state. We'd like the node i to turn `on` whenever certain conditions are met, which we can express as a Boolean function $f_i^{\texttt{on}}(x_i, x_k, x_l, x_m, \dots)$ of some nodes' states. To satisfy the finite/fading memory requirement, the `on` state must then revert back to the `off` state after some amount of time τ_i. Likewise, the node should turn `off` (or stay off[4]) whenever another condition $f_i^{\texttt{off}}$ is met. In other words, the node's output resembles a binary pulse of finite duration τ_i, a *monoflop*, encoded into the `start` and `stop` events t^i_{start} and t^i_{stop}.

I'll assume here that the duration τ_i is constant and the node is not re-triggerable, i.e. once triggered, it will stay `on` for exactly τ_i or until it is disabled by bad tokens, even if f_i is satisfied at a later point during this time-interval.[5] I'll also assume that there is an infinitesimal delay associated with the transmission of each event, which implies, for example, that a node that disables itself through negative feedback will generate a token with infinitesimal duration. In a clocked implementation, this translates to a minimal duration of one cycle.

The state of the entire system at any given point in time is thus determined by a single real number t_i per node i, the remaining duration of the current token if the node is currently in state $x_i = \texttt{on}$ (i.e. $t_i > 0$), or $t_i \leq 0$ if the node is in state $x_i = \texttt{off}$.

To implement $f_i^{\texttt{on}}$, we write this Boolean function in its conjunctive normal form:

$$f_i^{\texttt{on}} = \bigvee_{j \in O_i} \underbrace{\bigwedge_{k \in A_{i,j}} w_{i,j,k} x_k}_{=: g_{i,j}}, \tag{1}$$

where we use the sign $w_{i,j,k} \in \{-1, 1\}$ to represent logic negation ($-x_k \leftrightarrow \neg x_k$), and where O_i and $A_{i,j}$ index the *or*- and *and*-terms of $f_i^{\texttt{on}}$, respectively. If we introduce an intermediate term $g_{i,j}$ for each *or*-term, we see that our node i should turn on, if any of the intermediate terms $g_{i,j}$ evaluate to true. Let's first treat these intermediate terms, and then return to node i.

For each term $g_{i,j}$, we can introduce a "helper node" $h = h_{i,j}$ with state x_h that should turn `on` if and only if

$$g_{i,j} = \bigwedge_{k \in A_{i,j}} w_{i,j,k} x_k = \text{true}, \tag{2}$$

i.e. if the specific set of nodes $k \in A_{i,j}$ is in the respective states $w_{i,j,k}$. We can split these nodes further into two groups, the negated nodes $A^-_{i,j} = \{k \in A_{i,j} : w_{i,j,k} = -1\}$ and

[4] To be consistent, $f_i^{\texttt{off}} = \text{true}$ must imply $f_i^{\texttt{on}} = \text{false}$

[5] If $f_i^{\texttt{on}}$ is satisfied when the token is set to expire, a new one is generated that effectively extends the duration by another τ_i.

the non-negated nodes $A^+_{i,j} = A_{i,j} \setminus A^-_{i,j}$, so that equation 2 becomes:

$$g_{i,j} = (\bigwedge_{k \in A^+_{i,j}} x_k) \wedge (\bigwedge_{l \in A^-_{i,j}} \neg x_l) = (\bigwedge_{k \in A^+_{i,j}} x_k) \wedge (\neg \bigvee_{l \in A^-_{i,j}} x_l) \tag{3}$$

In words, node h should turn `on` if **all** nodes in the set $A^+_{i,j}$ are/turn `on` (the "good" tokens), and it should turn `off`, if **any** node in $A^-_{i,j}$ is/turns `on` (the "bad" tokens). To satisfy our locality requirement above, the node h needs to keep track of the state of all these nodes internally, in addition to its own state. Luckily, we only need to know *how many* nodes in $A^+_{i,j}$ and likewise in $A^-_{i,j}$ are `on`, i.e. we need to track just two variables, the "good" and the "bad token count":

$$c_h^{\text{good}} = |\{k \in A^+_{i,j} : x_k = \texttt{on}\}| \tag{4}$$

$$c_h^{\text{bad}} = |\{k \in A^-_{i,j} : x_k = \texttt{on}\}| \tag{5}$$

Using these two, equation 3 simplifies to just

$$g_{i,j} = (c_h^{\text{good}} = |A^+_{i,j}|) \wedge (c_h^{\text{bad}} = 0), \tag{6}$$

or, more generally:

$$g_{i,j} = (c_h^{\text{good}} \geq \theta_h^{\text{good}}) \wedge (c_h^{\text{bad}} < \theta_h^{\text{bad}}), \tag{7}$$

where θ_h^{good} and θ_h^{bad} are the "good" and "bad token threshold", two parameters of node h. As discussed above, node h keeps track of these two token counts by counting `start` and `stop` events from the nodes in $A_{i,j}$. By construction, node h then turns `on` whenever the condition $g_{i,j}$ becomes satisfied, and `off` whenever it ceases to be satisfied.

Now let's return to the original node i; it should turn `on` whenever any of the helper nodes h are on, i.e. equation 1 reduces to

$$f_i^{\texttt{on}} = \bigvee_{h \in \bar{O}_i} x_h \equiv c_i^{\text{good}} > 0, \tag{8}$$

where $\bar{O}_i$ is the new "good neighborhood" of node i, containing all the newly created helper nodes h. We can follow the same construction for deriving $f_i^{\texttt{off}}$ by introducing additional helper nodes.

Note that equation 8 is merely a special case of equation 7; in other words, any node should just turn `on` once its good token threshold has been reached or exceeded, and it should turn / stay `off`, once its bad token threshold has been exceeded. As the construction above shows, this allows us to construct arbitrarily complex Boolean conditions on when a node should (not) turn `on` or `off`, which is why I consider the TTT framework a simple yet *complete event-driven logic*.

III. Modeling examples

With these formalities out of the way, let's have a look what this framework can do. In the following, I will show a few different examples of event-based systems and building blocks that can be constructed from simple circuits of several TTT nodes.

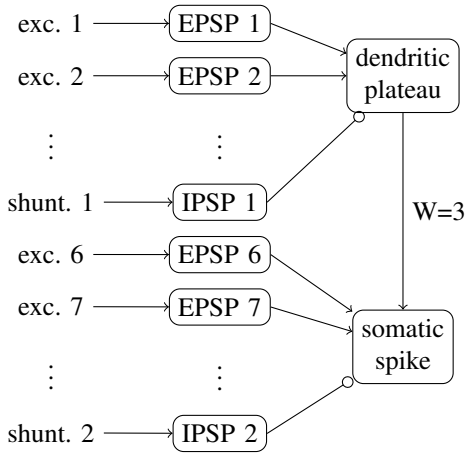


Fig. 2: Neuron circuit constructed from TTT blocks.

A. Spiking neurons (with active dendrites)

If you are familiar with SNNs you may have noticed that the construction above is very similar to a simple spiking neuron, where the good incoming tokens represent excitatory post-synaptic potentials (EPSPs), and the bad incoming tokens represent (shunting) inhibitory post-synaptic potentials (sIPSPs). The main differences are that the TTT node sums good and bad tokens separately, rather than in a single membrane potential, and it can generate a potentially long lasting token (or pulse), which it can also prematurely terminate. Typical SNN neuron can only output an idealized spike event with infinitesimally short duration, but the ability to generate longer pulses comes in handy if we want to model neurons with long-lasting dendritic plateau potentials according to [2].

Fig. 2 shows how such a neuron with a somatic and a dendritic compartment and 10 excitatory and 2 inhibitory synaptic connections can be implemented in the TTT framework. Here, each pulse with its own time-constant, i.e. each EPSP, IPSP, dendritic plateau potential and somatic spike, is implemented by one node with correspondingly chosen token duration τ_i. The excitatory synpases target the good token place of the respective compartment, whereas the inhibitory synapses target the bad token place, each with a weight of 1. To model the strong influence of a dendritic plateau potential on the somatic compartment, we assign a larger positive weight (3) between them. The good and bad token thresholds are set to 3 and 0 for the dendritic compartment, and to 6 and 0 for the somatic compartment.

If we simulate this system of TTT nodes and stimulate it with randomly timed spikes, represented by incoming tokens with short duration $\tau_{\text{in}} = 10\,\text{ms}$, we see that it behaves like a coincidence detecting neuron. Fig. 3 shows the time-course of such a simulation, where the lower two panels track the good token count in both compartments - an analog to membrane potentials. Here, strong coincident input to the dendritic compartment triggers plateau potentials at around $300\,\text{ms}$, $600\,\text{ms}$ and $800\,\text{ms}$. Each plateau lasts for $\tau = 100\,\text{ms}$ unless, as in the second case, an IPSP interrupts it.

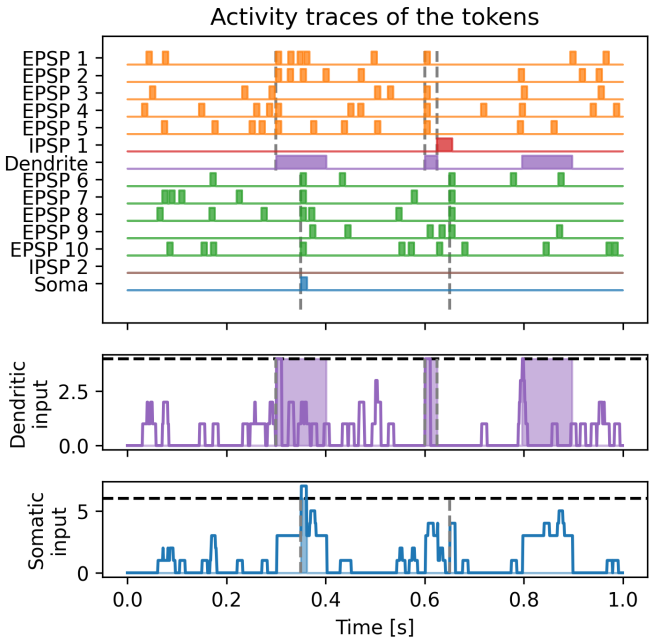
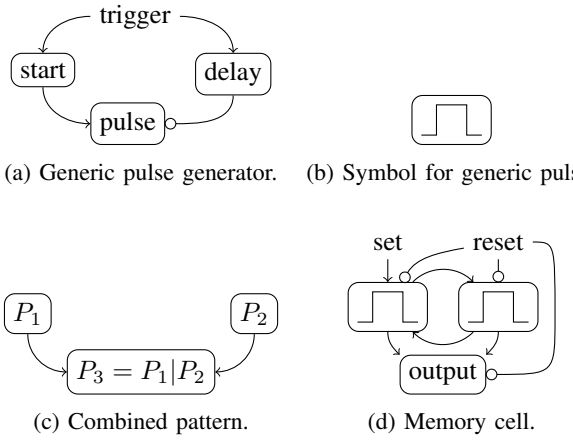


Fig. 3: Simulated behavior of the neuron from Fig. 2



(a) Generic pulse generator. (b) Symbol for generic pulse.

(c) Combined pattern. (d) Memory cell.

Fig. 4: Useful building blocks constructed from TTT blocks.

During a dendritic plateau, e.g. at 350 ms, additional coincident input to the soma can trigger the neuron to spike, whereas synaptic input alone is insufficient, as we can see at 650 ms. This neuron hence detects a sequence of a coincident volley of spikes targeting the dendritic compartment, followed within 100 ms by another volley of coincident spikes targeting the soma.

B. Arbitrary pattern detection & generation

Besides the special case of SNNs, the TTT framework is generic enough to model arbitrary event-driven computations. To demonstrate this, I will show how various useful features and building blocks can be implemented (see Fig. 4), starting with pattern generators and detectors.

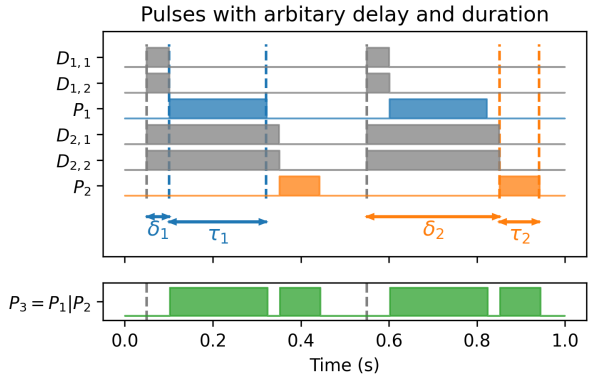


Fig. 5: Simulated behavior of the pulse generator from Fig. 4 (a) and (c).

1) Pattern generation: To generate arbitrary temporal patterns in response to some trigger event, we first need to be able to generate individual pulses with specific delay and duration. As shown in Fig. 4(a), this can be realized by three nodes, *start*, *delay* and *pulse*. The *start* and *delay* nodes are both triggered by the input, and have durations $\tau_{\text{delay}} = \delta_1$ and $\tau_{\text{start}} > \tau_{\text{delay}}$. The *pulse* node has duration τ_1. Once triggered, the bad *delay* token will inhibit the *pulse* node until it expires after δ_1, at which point the good *start* token turns *pulse* `on`. This results in a pulse that starts with a delay δ_1 and lasts for a duration τ_1. If we want to *dynamically* control the duration of a token, we can use multiple good and bad tokens of fixed duration, and combine them depending on the system's state to prolong or shorten the token.

We can combine two or more of these pulse generators into an arbitrary pattern generator, e.g. by connecting them to another node that can be triggered by one of these pulse generators, as show in Fig.4(c). If we set this node's token duration short, it will remain `on` (or continuously re-trigger) during any of the pulses, effectively computing the disjunction of the pulses. Fig. 5 shows an example of a pattern generator that produces a sequence of two pulses with fixed delays δ_1, δ_2 and durations τ_1, τ_2.

2) Pattern detection: We can also use pattern generators to *detect* specific patterns. To detect a certain temporal sequence of multiple events, we can generate a token with specifically chosen delay and duration for each of them, such that all of these tokens coincide whenever the sequence is presented. A single node with a good token threshold equal to the number of events in the pattern will then only fire if the full pattern was present. Additional bad tokens can be introduced to suppress the detection of confounding patterns.

C. Memory cells

To perform useful computations, our system should be able to preserve information indefinitely in some form of memory cell. Due to our fading memory requirement, this must be implemented by a circuit of multiple nodes. Here we can build on the pulse generators introduced previously and construct

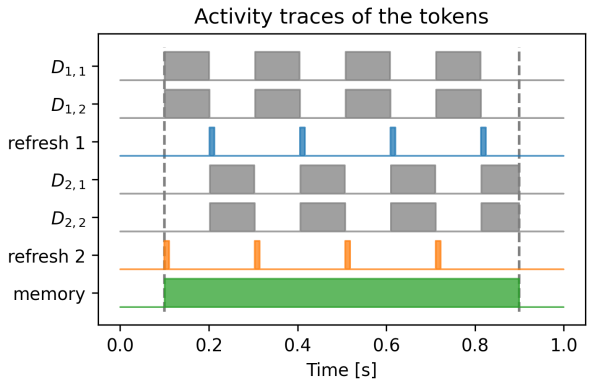


Fig. 6: Simulated behavior of the memory cell from Fig. 4 (d).

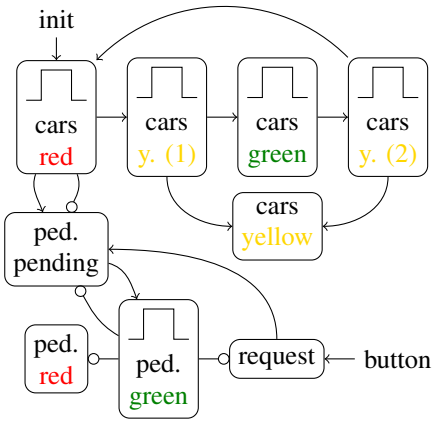


Fig. 7: Traffic light example.

something that resembles an event-based flip-flop: two pulse generators with long delay δ and short duration τ activate each other through mutual positive connections. One additional node with duration δ represents the cell's output; it is refreshed periodically by the two pulse generators. The memory cell can be *set* by an incoming token that triggers one of the pulse generators, and *reset* by another event that interrupts all ongoing tokens. Fig. 6 shows a simulation of such a memory cell, which is set at $100\,\mathrm{ms}$ and reset at $900\,\mathrm{ms}$.

D. Building complex systems

The building blocks described above can be used to construct complex systems akin to (timed) FSMs and Time Petri Nets. To this end, we can use each node to represent a single bit in the state vector of a complex system. The conditions for transitioning from one global state to another can then be broken down into local conditions for each node, which can be implemented by introducing helper nodes as derived above. The ability to generate arbitrary temporal patterns endows this system with a powerful representation of time.

To give an example of a more involved system, let's consider a traffic light with a pedestrian crossing. The cars' traffic light should cycle through four phases (red, yellow, green, yellow), each with its own duration. If (and only if!) a pedestrian has

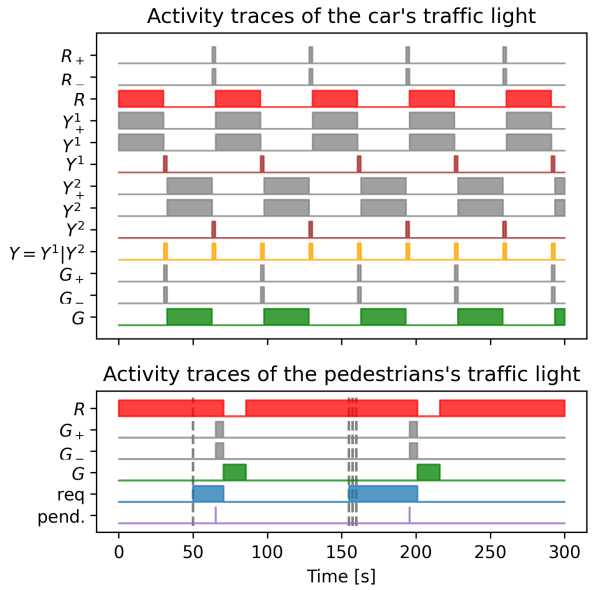


Fig. 8: Simulated behavior of the traffic-light state-machine from Fig. 7.

requested to cross, the pedestrians' traffic light should turn green for some fixed interval in the middle of the cars' red phase, and then revert to red. A pedestrian can request to cross at any time by pushing a button, but this has no effect if the pedestrian light is green already, and a pending request only takes effect once the next red phase starts.

Fig. 7 shows how this system can be implemented using five pulse generators, one for each phase with fixed duration, and four additional nodes. To keep the pedestrian traffic light red whenever it is not green, we set the corresponding good token threshold to zero, and we use one node to combine the two different yellow phases. The pedestrians' requests are arbitrated by two nodes, one of which remembers that a request has been made, and another that decides whether to grant the request in the current red cycle. This last node can only be triggered at the beginning of the cars' red phase if a request was made previously, and it is disabled immediately after, so that any request made during the cars' red phase will only take effect in the next cycle. Finally, the pedestrian green light clears any outstanding requests. Fig. 8 shows a simulation of the traffic light system. As we can see, the traffic light correctly cycles through the different phases, and a request to cross at $50\,\mathrm{s}$ only takes effect at $60\,\mathrm{s}$ during the subsequent cars' red phase. Repeated requests during or after the cars' red phase around $155\,\mathrm{s}$ take effect in the subsequent red phase at $210\,\mathrm{s}$.

This non-trivial example hopefully demonstrates that the TTT framework, as simple as it may be, is able to model complex event-driven system by composing instances of a single basic component, the TTT node with its good and bad token counter.

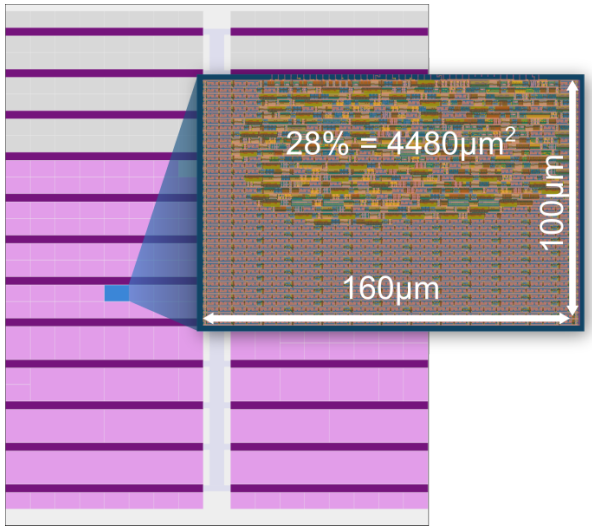


Fig. 9: Synthesized layout and its placement on the TinyTapeout chip.

IV. VLSI IMPLEMENTATION

I implemented a single core as shown in Fig. 1 in Verilog, synthesized it using the open-source workflow for TinyTapeout [1], and submitted it for fabrication in the $130\,\mathrm{nm}$ SkyWater process as part of TinyTapeout05 [29] multi-project wafer. Fig. 9 shows the generated layout and its placement on the chip. Its netlist comprises 558 standard cells (excluding fill and tap cells), and at $4480\,\mu\mathrm{m}^2$ it utilizes only 28% of the available $100\,\mu\mathrm{m} \times 160\,\mu\mathrm{m}$ area.

The core uses $8\,\mathrm{bit}$ token counters and count-down timers, and has programmable thresholds as well as a programmable duration. To test networks composed of multiple cores, I will use a time-multiplexing scheme outlined in the code repository.

V. CONCLUSION

A. Extensions

Many extensions of this model are possible and perhaps worth considering. For example, rather than assuming infinitesimal delays, we could consider finite or even parameterized delays δ_i. Currently, a delay can be implemented via a generic pulse generator as shown in Fig. 4(a), but this requires three nodes; including a (programmable) delay in each node would only require one additional count-down timer. We could also make the nodes re-triggerable, which would change the dynamics in subtle ways. We could also increase the nodes' complexity, e.g. the duration of each token could be stochastic and / or dependent on internal token counts, or we could add more internal places with their own token counters to make it easier to implement certain conditions f_i.

B. So what?

What role could the TTT framework play for future neuromorphic research? I personally intend to use it to study event-driven computing in general, and dendritic computation specifically. For these purposes, the framework is not only a good formalism to define and analyze networks, but it also provides a constructive toolkit to build up increasingly complex micro- and then macro-circuits that can reliably produce certain behavior. Like in conventional logic, I believe it is possible to then invert this process and *synthesize* event-driven systems from a high-level description of their behavior.

Due to its inherently parallel construction, this framework (unlike Petri Nets) might also prove to be a good basis for building efficient *simulators* for event driven systems. But more interestingly, from a hardware perspective, I see the proposed TTT node as an attempt to define a kind of *logic gate* for event-driven computing, which can be easily implemented in highly integrated digital CMOS circuits. Future work could leverage analog computing to further reduce the power consumption or required area, or, by using programmable routing schemes that have been developed by the neuromorphic engineering community, this concept could be scaled-up to a kind of event-driven field programmable gate array ("evFPGA"), which would greatly lower the entry barrier for designing and analyzing event-driven computing systems.

VI. CODE AVAILABILITY

All code used here, including a simulator, code to generate all figures, the Verilog code used for synthesizing the hardware implementation and corresponding test-benches can be found in a public repository at https://github.com/jleugeri/tnt-ticktocktokens.

ACKNOWLEDGMENT

The original idea for this concept goes back to joint work with Pascal Nieters on a simple event-based model of neural computation. I developed the hardware implementation for the TinyNeuromorphicTapeout (TNT) project initiated at the 2023 Telluride Neuromorphic Cognition Engineering Workshop. I'd like to thank Jason Eshraghian and Peng Zhou for organizing this topic area and financing my slot on the tapeout, as well as Renaldas Zioma for helpful discussions. The TNT project in turn is based on TinyTapeout, so special thanks go to Matt Venn and the TinyTapeout community for their efforts to democratize VLSI design.

REFERENCES

[1] Tiny tapeout 05. [Online]. Available: https://tinytapeout.com/runs/tt05/

[2] J. Leugering, P. Nieters, and G. Pipa, "Dendritic plateau potentials can process spike sequences across multiple time-scales," vol. 2. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fcogn.2023.1044216

[3] J. Leugering, "Modeling and analyzing neuromorphic SNNs as discrete event systems," in *Neuro-Inspired Computational Elements Conference*. Virtual Event USA: ACM, Mar. 2022, pp. 61–62. [Online]. Available: https://dl.acm.org/doi/10.1145/3517343.3517362

[4] ——, "Making spiking neurons more succinct with multi-compartment models," in *Proceedings of the Neuro-inspired Computational Elements Workshop*. ACM, pp. 1–6. [Online]. Available: https://dl.acm.org/doi/10.1145/3381755.3381763

[5] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Springer International Publishing, 2018. [Online]. Available: http://link.springer.com/10.1007/978-3-319-10575-8

[6] R. Alur and D. L. Dill, "A theory of timed automata," vol. 126, no. 2, pp. 183–235. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0304397594900108

[7] L. Popova-Zeugmann, *Time and Petri Nets*. Berlin, Heidelberg: Springer, 2013. [Online]. Available: https://link.springer.com/10.1007/978-3-642-41115-1

[8] G. Ciatto, E. de Maria, and C. Di Giusto, "Modeling Third Generation Neural Networks as Timed Automata and verifying their behavior through Temporal Logic," Université Côte d'Azur, CNRS, I3S, France, Research Report, Feb. 2017. [Online]. Available: https://hal.science/hal-01473941

[9] E. De Maria, C. Di Giusto, and G. Ciatto, "Formal Validation of Neural Networks as Timed Automata," in *Proceedings of the 8th International Conference on Computational Systems-Biology and Bioinformatics*. Nha Trang City Viet Nam: ACM, Dec. 2017, pp. 15–22. [Online]. Available: https://dl.acm.org/doi/10.1145/3156346.3156350

[10] E. De Maria, C. Di Giusto, and L. Laversa, "Spiking neural networks modelled as timed automata: with parameter learning," *Natural Computing*, vol. 19, no. 1, pp. 135–155, Mar. 2020. [Online]. Available: http://link.springer.com/10.1007/s11047-019-09727-9

[11] W. Maass and H. Markram, "On the computational power of circuits of spiking neurons," *Journal of Computer and System Sciences*, vol. 69, no. 4, pp. 593–616, Dec. 2004. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0022000004000406

[12] S.-C. Liu, Ed., *Event-based neuromorphic systems*. The Atrium, Southern Gate, Chichester, West Sussex, United Kingdom: John Wiley & Sons, Ltd, 2015.

[13] L. Bonilla, J. Gautrais, S. Thorpe, and T. Masquelier, "Analyzing time-to-first-spike coding schemes: A theoretical approach," *Frontiers in Neuroscience*, vol. 16, 2022. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fnins.2022.971937

[14] S. Thorpe and J. Gautrais, "Rank Order Coding," in *Computational Neuroscience: Trends in Research, 1998*, J. M. Bower, Ed. Boston, MA: Springer US, 1998, pp. 113–118. [Online]. Available: https://doi.org/10.1007/978-1-4615-4831-7_19

[15] A. A. Lazar, E. K. Simonyi, and L. T. Toth, "Time encoding of bandlimited signals, an overview," in *Proceedings of the Conference on Telecommunication Systems, Modeling and Analysis*, Nov 2005.

[16] A. Taherkhani, A. Belatreche, Y. Li, G. Cosma, L. P. Maguire, and T. M. McGinnity, "A review of learning in biologically plausible spiking neural networks," vol. 122, pp. 253–272. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608019303181

[17] J. V. Neumann, *The Computer and the Brain*. New Haven: Yale University Press, 1958.

[18] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

[19] M. Minsky and S. A. Papert, *Perceptrons: an introduction to computational geometry*, 2nd ed. Cambridge/Mass.: The MIT Press, 1972.

[20] E. Mizraji and J. Lin, "Logic in a Dynamic Brain," *Bulletin of Mathematical Biology*, vol. 73, no. 2, pp. 373–397, Feb. 2011. [Online]. Available: https://doi.org/10.1007/s11538-010-9561-0

[21] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[22] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Cham: Springer International Publishing, 2021. [Online]. Available: https://link.springer.com/10.1007/978-3-030-72274-6

[23] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978. [Online]. Available: https://dl.acm.org/doi/10.1145/359576.359585

[24] D. Brand and P. Zafiropulo, "On Communicating Finite-State Machines," *Journal of the ACM*, vol. 30, no. 2, pp. 323–342, Apr. 1983. [Online]. Available: https://dl.acm.org/doi/10.1145/322374.322380

[25] A. J. Martin, "Compiling communicating processes into delay-insensitive VLSI circuits," vol. 1, no. 4, pp. 226–234. [Online]. Available: https://doi.org/10.1007/BF01660034

[26] A. Madhavan, M. W. Daniels, and M. D. Stiles, "Temporal State Machines: Using Temporal Memory to Stitch Time-based Graph Computations," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 17, no. 3, pp. 28:1–28:27, May 2021. [Online]. Available: https://doi.org/10.1145/3451214

[27] X. Lagorce and R. Benosman, "STICK: Spike Time Interval Computational Kernel, A Framework for General Purpose Computation using Neurons, Precise Timing, Delays, and Synchrony," Jul. 2015. [Online]. Available: https://arxiv.org/abs/1507.06222v1

[28] J. V. Monaco, M. M. Vindiola, and R. Benosman, "STEAM: Spike Time Encoded Addressable Memory." [Online]. Available: https://vmonaco.com/papers/Poster-%20STEAM-%20Spike%20Time%20Encoded%20Addressable%20Memory.pdf

[29] Tiny tapeout. [Online]. Available: https://tinytapeout.com/