PyLSE: A Pulse-Transfer Level Language for Superconductor Electronics

Michael Christensen
Department of Computer Science
UC Santa Barbara
USA
mchristensen@cs.ucsb.edu

Georgios Tzimpragos
Department of Computer Science
UC Santa Barbara
USA
gtzimpragos@cs.ucsb.edu

Harlan Kringen
Department of Computer Science
UC Santa Barbara
USA
kringen@cs.ucsb.edu

Jennifer Volk
Department of Electrical and Computer Engineering
UC Santa Barbara
USA
jevolk@ucsb.edu

Timothy Sherwood
Department of Computer Science
UC Santa Barbara
USA
sherwood@cs.ucsb.edu

Ben Hardekopf
Department of Computer Science
UC Santa Barbara
USA
benh@cs.ucsb.edu

Abstract

Superconductor electronics (SCE) run at hundreds of GHz and consume only a fraction of the dynamic power of CMOS, but are naturally pulse-based, and operate on impulses with picosecond widths. The transiency of these operations necessitates using logic cells that are inherently stateful. Adopting stateful gates, however, implies an entire reconstruction of the design, simulation, and verification stack. Though challenging, this unique opportunity allows us to build a design framework from the ground up using fundamental principles of programming language design. To this end, we propose PyLSE, an embedded pulse-transfer level language for superconductor electronics. We define PyLSE through formal semantics based on transition systems, and build a framework around them to simulate and analyze SCE cells digitally. To demonstrate its features, we verify its results by model checking in UPPAAL, and compare its complexity and timing against a set of cells designed as analog circuit schematics and simulated in Cadence.

CCS Concepts: • **Hardware** → **Hardware description languages and compilation**; *Emerging technologies*; • **Theory of computation** → *Timed and hybrid models.*

Keywords: superconductor electronics, hardware description language, timed automata

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523438

ACM Reference Format:
Michael Christensen, Georgios Tzimpragos, Harlan Kringen, Jennifer Volk, Timothy Sherwood, and Ben Hardekopf. 2022. PyLSE: A Pulse-Transfer Level Language for Superconductor Electronics. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523438

1 Introduction

Superconductor electronics (SCE) are a promising emerging technology for the post-Moore era — especially for large-scale [23], machine learning [26, 52], and quantum [22, 34] computing systems — due to their energy-efficient interconnects and sub-attojoule ultra-high-speed switching [24]. However, the physical properties that make SCE so promising also make them difficult to design for. In particular, SCE use a *pulse-based*, rather than voltage level-based, information encoding. This, along with the *stateful* nature of superconducting cells [47] and the lack of a uniformly agreed-upon efficient translation from design to implementation, makes it necessary to develop unique logic gates and design rules [11, 52, 54].

The primary question we seek to answer in this paper is: *what is a suitable abstraction for precisely defining the functional and timing behavior of SCE designs*? Our solution is to completely depart from existing hardware description languages (HDLs) and instead take a bottom-up approach to build a new Python [41] embedded domain-specific language (DSL) called PyLSE (**Py**thon **L**anguage for **S**uperconductor **E**lectronics). We argue that PyLSE is well-tailored to the unique needs of SCE, making it easier to create and compose cells into correct and scalable systems.

Inspired by the theory of automata [25], we propose a custom finite state machine (FSM) abstraction, which we call a PyLSE Machine and which forms the core our PyLSE language. This FSM abstraction allows the description of

(a) CMOS: Steady voltage levels encode information; thus, wires are considered *stateful* and the gates *stateless.*

(b) SFQ: Transient pulses encode information; thus, wires are considered *stateless* and the gates *stateful.*

Figure 1. Information representation in CMOS and SFQ.

the functional and timing behavior of SCE cells without the complex and error-prone conditional assignments commonly found in state-of-the-art approaches [37] Through this abstraction, we also develop a new link between SCE and the theory of Timed Automata (TA) [2], which enables the integration of PyLSE with modern formal verification tools like the UPPAAL model checker [7]. Overall, the main contributions of this paper are:

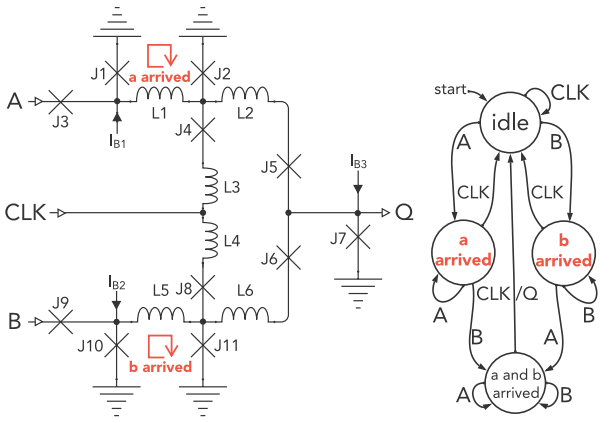
- We create the PyLSE Machine, a language abstraction for the formalization of the functional and timing semantics of pulse-based circuits (Section 3).
- We create PyLSE, a lightweight transition system-based Python DSL for the rapid prototyping of pulse processing systems, modeled as networks of PyLSE Machines (Section 4).
- We automate the translation of PyLSE Machines to Timed Automata (Section 4).
- We build a multi-level framework for the simulation and analysis of PyLSE Machine systems, which also allows for the integration of abstract behavioral software models, fostering agile development (Section 4).
- We evaluate PyLSE's capabilities through a series of comparisons with state-of-the-art approaches, dynamic checks of SCE designs with stochastic timing behaviors, and formal verification using UPPAAL (Section 5).

2 Defining Computation on Pulses

2.1 Functional Behavior

Superconductor electronics exploit the unique properties of superconductivity [14, 27] to perform computation through the carefully orchestrated consumption and emission of individual packets of magnetic energy, which manifest as single flux quanta (SFQ) pulses [31]. While the quantum nature of such flux exchange is central to the device operation, the computation performed is strictly classical. Information moves between logic elements in the same "feed-forward" way as traditional digital logic. However, the use of picosecond-scale SFQ *pulses*, rather than the sustained voltage levels of CMOS (see Figure 1a), has myriad downstream effects. Most notably, SFQ cells[1] must be designed to "remember" that a particular input has arrived (see Figure 1b).

[1]We use "cell," "gate," and "element" interchangeably throughout.



(a) Schematic **(b)** Mealy machine

Figure 2. Schematic and Mealy machine description of a Synchronous And Element. Labels `a arrived` and `b arrived` in the schematic are locations of superconducting loops holding state and roughly correspond to states in the Mealy machine.

The SCE community has traditionally relied on low-level analog models for the design and analysis of basic SCE cells. However, the growing interest of digital designers in SCE means that there is an increased need for new abstractions that are more suitable for scaling SCE system design and analysis. One abstraction commonly used to explain the stateful behavior of SCE cells is the Mealy machine [35]. Mealy machines have been used extensively to model SCE cells [19, 31, 52–54, 60]. For example, Figure 2b describes the functionality of a Synchronous And Element without the low-level circuit details of Figure 2a.

2.2 Timing Behavior

A depiction of the Synchronous And Element waveform is shown in Figure 3. Its hold and setup times, as well as its propagation delays, are defined similar to convention. Namely, setup time and hold time are defined as the intervals before and after the clock in which no pulses should arrive, respectively [29, 30]. In this figure, event ① indicates a hold time violation by input A while event ② indicates a setup time violation by Input B. Each of these events may cause a pulse to be dropped or the cell to enter a metastable state. Propagation delay measures the time between the arrival of a pulse that will trigger an output and the generation of the actual output pulse – in the case of a Synchronous And Element, it is measured from the arrival of the clock pulse to the output pulse (event ③).

Because Mealy machines lack an explicit notion of time, they fall short when constraints on the relative arrival times of inputs must be part of the functional description. These and other timing restrictions need to be carefully thought through and should be captured as early in the design process as possible. A good language abstraction must therefore treat

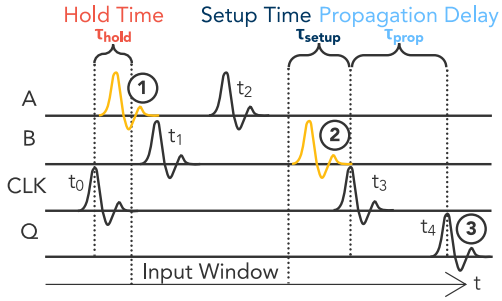


Figure 3. Waveform for the Synchronous And Element showing the timing constraints that must be met for correct operation. Pulses that arrive during the hold time ① or setup time ② are erroneous. Assuming these timing violations do not occur, a pulse is produced some propagation delay ③ after a clock pulse.

time as a central, *first class concern* and provide mechanisms for (1) easily defining timing constraints and (2) verifying the absence of violations in the system.

3 Overview of the PyLSE Machine

There are four key pieces of information that must be captured by any new abstraction for superconductor technology:

1. The state transition time;
2. The prioritization of simultaneous input signals;
3. The propagation time per cell; *and*
4. The time windows for valid inputs.

We use the Mealy machine as a base for this new *PyLSE Machine* abstraction and augment its edges to cover the timing properties and constraints discussed in Section 2. These augmented edges consist of three parts: the **Trigger** (which includes input, priority, and transition times), the **Firing Outputs** (which associates each output with its firing delay), and **Past Constraints** (which is a way to specify the legal relative arrival times of wires); the details of each are found in Figure 4. The machine must be *fully-specified* such that for all states, all inputs are associated with edges.

To show how the proposed extension transforms a Mealy machine, like that shown in Figure 2b, into a PyLSE Machine, we use the Synchronous And Element[2] as a running example. More specifically, to highlight how the features of Figure 5 can be used to express the setup and hold time constraints and propagation delay of this cell, we dissect the edge (colored in gold) that connects the `a and b arrived` state to `idle` ($\text{CLK}^0_{\tau_{hold}}/\{Q_{\tau_{prop}}\}/\{*_{\tau_{setup}}\}$).

[2]The Synchronous And Element assumes an RSFQ [31] encoding in which the presence of a pulse on an output between clock pulses encodes a 1 and the absence of a pulse encodes a 0, although other pulse-to-value mappings such as temporal [52] and xSFQ [54] are possible and work in PyLSE.

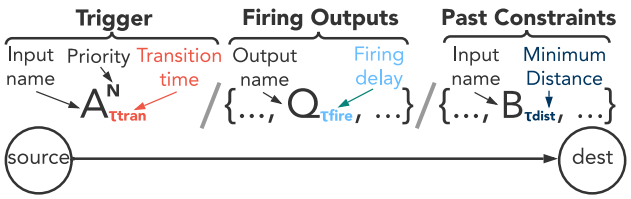


Figure 4. Anatomy of a PyLSE Machine transition. The arrival of an input pulse on wire `A` **Trigger**s the transition from the *source* to *dest* state. This transition has priority N over other simultaneously-triggered transitions originating from *source* and takes τ_{tran} time to complete; during this period, receiving any inputs is illegal. A pulse for each output `Q` in the **Firing Outputs** set appears on their associated output wire some τ_{fire} time units later. Finally, according to the **Past Constraints**, if it's been less than τ_{dist} since the last time an input `B` was received during a previous transition, it is an error. $\text{A}^N_{\tau_{tran}}$ is shorthand for $\text{A}^N_{\tau_{tran}}/\varnothing/\varnothing$.

Transition Time. The **Trigger** portion indicates that a state transition will occur upon the arrival of `CLK`. The amount of time required for this transition to complete is τ_{hold} time units. Moreover, to model the *hold time* constraint of Figure 3, we set $\tau_{tran} := \tau_{hold}$ and therefore consider the arrival of any other input pulse during this transitionary period illegal.

Priorities: The **Trigger** portion also indicates the priority among the edges departing from the same node. For example, the highlighted edge has a priority of `0`. This implies that even if the machine received `A`, `B`, and `CLK` simultaneously, it will always handle the transition associated with `CLK` first. Once this transition completes and the machine settles into the `idle` state, an arbitrary choice between `A` and `B` is made, because both of them have the same labeled priority of 1. We note that although it is practically impossible to arrange for SFQ pulses to purposefully arrive "simultaneously," it is not uncommon to consider models of gates with coarse timing.

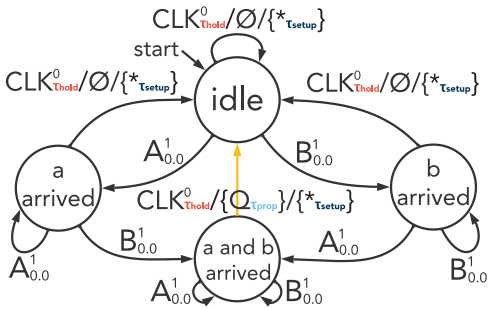


Figure 5. PyLSE Machine for the Synchronous And Element. Using transition time, we can model the hold time τ_{hold} and using the past constraints, we can model the τ_{setup} time. Firing delay directly models the propagation delay τ_{prop}.

In this case, priorities let the designer identify and explicitly handle cases of simultaneous arrivals deterministically.

Multi-Output: To model arbitrary SFQ cells, we should also be able to associate a *set* of outputs with each edge and define their timing; the **Firing Outputs** portion does just that. As can be seen in the provided example, the singleton set $\{Q_{\tau_{\texttt{prop}}}\}$ indicates that an output pulse will be emitted during this state transition. The time that it will take for this pulse to appear is $\tau_{\texttt{prop}}$ time units. Therefore, we can use the edge's firing delay to model the cell's *propagation delay*; e.g., by setting $\tau_{\texttt{fire}} \coloneqq \tau_{\texttt{prop}}$.

Constraints on Past: The **Past Constraints** portion is used to model the *setup time* constraint; e.g., by setting $\tau_{\texttt{dist}} \coloneqq \tau_{\texttt{setup}}$. In the provided example, any input pulse (indicated by the $*$) that appears within $\tau_{\texttt{setup}}$ time units after the arrival of CLK is considered illegal.

3.1 Formalization of the PyLSE Machine

In this section, we define PyLSE Machines, their semantics, and how they interact in larger designs.

Definition 3.1 (PyLSE Machine)**.** A finite state machine with timed prioritized transitions, an output set, and past constraints, which we call a *PyLSE Machine*, is a tuple $M = \langle Q, q_{\texttt{init}}, \Sigma, \Lambda, \delta, \mu, \theta \rangle$, where

$$
\begin{aligned}
(q \in) Q &\text{ is a set of states} \\
q_{\texttt{init}} \in Q &\text{ is the initial state} \\
(\sigma \in) \Sigma &\text{ is a set of input symbols} \\
(\lambda \in) \Lambda &\text{ is a set of output symbols} \\
\delta : Q \times \Sigma \to Q \times \mathbb{N} \times \mathbb{R} &\text{ is the transition function} \\
\mu : Q \times \Sigma \to \mathcal{P}(\Lambda \times \mathbb{R}) &\text{ is the output function} \\
\theta : Q \times \Sigma \to \mathcal{P}(\Sigma \times \mathbb{R}) &\text{ is the past constraints function}
\end{aligned}
$$

We write $M.\Sigma$ to extract Σ, and likewise $M.\Lambda$ for Λ.

The first three domains — Q, Σ, and Λ — are similar to a typical Mealy machine definition. The transition function δ maps a state and input symbol to (1) the next state it should transition to, (2) a natural number corresponding to the priority of that transition, and (3) a real number corresponding to the physical time it takes to complete. The output function, μ, maps tuples of states and inputs to *sets* of tuples consisting of output symbols and the time it takes for them to appear (i.e. a firing delay). The past constraints function θ maps the current state and input to a input–real number tuple. This tuple indicates a precondition for the given transition to be allowed to proceed (specifically, the setup time constraint).

The transition semantics of our PyLSE Machine is found in Figure 6. To define the semantics, we use a configuration $\kappa \in \mathrm{K} = Q \times \mathbb{R} \times (\Sigma \to \mathbb{R})$, parameterized over a current state $q \in Q$, a real-valued time $\tau_{\texttt{done}}$, and a mapping $\Theta : \Sigma \to \mathbb{R}$ that associates each input with the last time it was seen. This is written as $\kappa_{\langle q, \tau_{\texttt{done}}, \Theta \rangle}$, with the $\tau_{\texttt{done}}$ being used to represent the end of the unstable period during which time the machine is transitioning. The initial configuration is $\kappa^M_{init} = \kappa_{\langle q_{\texttt{init}}, 0, \{\sigma \mapsto -\infty \mid \sigma \in M.\Sigma\} \rangle}$.

Transition Relation. Given the current configuration $\kappa_{\langle q_{\texttt{curr}}, \tau_{\texttt{done}}, \Theta \rangle}$, the Transition Relation is interpreted as follows. If the machine receives an input σ at time $\tau_{\texttt{arr}}$ and it has been long enough to have finished entering state $q_{\texttt{curr}}$ (i.e. $\tau_{\texttt{arr}} \geq \tau_{\texttt{done}}$), it proceeds to a new configuration $\kappa_{\langle q_{\texttt{next}}, \tau'_{\texttt{done}}, \Theta' \rangle}$. It does so by remembering (1) the next state $q_{\texttt{next}}$, (2) the time at which the new transition should be completed $\tau'_{\texttt{done}} = \tau_{\texttt{tran}} + \tau_{\texttt{arr}}$, and (3) the time it saw this current input, via $\Theta' = \Theta[\sigma \mapsto \tau_{\texttt{arr}}]$ (see Normal-κ). Otherwise, if it is not yet ready to receive inputs because $\tau_{\texttt{arr}} < \tau_{\texttt{done}}$ (see Error-κ Tran) or because any input σ' was received less than $\theta(q, \sigma') + \tau_{\texttt{dist}}$ ago (see Error-κ Cons), it proceeds to the special $q_{\texttt{err}}$ state. $q_{\texttt{err}}$ is the target state of any transition whose timing conditions can't be satisfied.

Dispatch and Trace Relations. The Dispatch Relation enables the machine to continue processing inputs. It works by retrieving the highest priority transition that leaves $q_{\texttt{curr}}$ for all the inputs σ in the set of simultaneous inputs $\vec{\sigma}$ arriving at $\tau_{\texttt{arr}}$. It chooses one nondeterministically if multiple candidate transitions have the same priority. The Trace Relation is used to determine the outputs that result from running the Dispatch Relation over the entirety of the inputs.

3.2 Formalizing a Network of PyLSE Machines

While each individual PyLSE Machine models a particular type of SCE cell, a network of communicating PyLSE Machines models a larger design.

Definition 3.2 (Network Domain of PyLSE Machines)**.** A network of PyLSE Machines, which we call a *circuit*, is a tuple $C = \langle \vec{M}, \vec{w}, \Sigma, \Lambda \rangle$ composed of a set of PyLSE Machines $\vec{M}$ (accessed as $C.machines$), a set of connective wires $\vec{w}$ (accessed as $C.wires$), and a set of circuit inputs $C.\Sigma$ and outputs $C.\Lambda$. A wire is a tuple $w = \langle \alpha, \beta \rangle$ such that $\alpha \in M'.\Lambda \bigcup C.\Sigma$ and $\beta \in M''.\Sigma \bigcup C.\Lambda$ for some $M', M'' \in \vec{M}$.

Network Relation. The Network Relation of Figure 6 shows the semantics of how a sequence of externally derived time-tagged pulses ts propagate through the network. We define an initial circuit configuration κ^C_{init}, composed of (1) all individual PyLSE Machine initial configurations $\vec{\kappa}$ and (2) a list of input pulses ps tagged with the wires where they are headed, i.e. $\kappa^C_{init} = \langle \vec{\kappa}, ps \rangle$, where $\vec{\kappa} = \{\kappa^M_{init} \mid M \in C.machines\}$ and $ps = \{\langle \sigma', \tau_{\texttt{arr}} \rangle \mid \langle \sigma, \tau_{\texttt{arr}} \rangle \in ts \wedge \langle \sigma, \sigma' \rangle \in C.wires\}$. The network proceeds until there is no more work to do, such that $\langle \kappa^C_{init}, ps \rangle \twoheadrightarrow_{net} \langle \kappa^{C'}, ps' \rangle$. In other words, all pending pulses in ps are directed toward the circuit output.

Transition Relation $\boxed{\rightarrow_{tran} \subseteq \mathrm{K} \times \Sigma \times \mathbb{R} \times (\mathrm{K} \cup \{q_{\mathtt{err}}\})}$

$$\frac{\langle q_{\mathtt{next}}, \_, \tau_{\mathtt{tran}}\rangle = \delta(q_{\mathtt{curr}}, \sigma) \quad \tau_{\mathtt{arr}} \geq \tau_{\mathtt{done}} \quad \forall \langle \sigma', \tau_{\mathtt{dist}}\rangle \in \theta(q_{\mathtt{curr}}, \sigma), \tau_{\mathtt{arr}} \geq \Theta[\sigma'] + \tau_{\mathtt{dist}}}{\kappa_{\langle q_{\mathtt{curr}}, \tau_{\mathtt{done}}, \Theta\rangle} \xrightarrow{\langle \sigma, \tau_{\mathtt{arr}}\rangle}_{tran} \kappa_{\langle q_{\mathtt{next}}, \tau_{\mathtt{tran}}+\tau_{\mathtt{arr}}, \Theta[\sigma \mapsto \tau_{\mathtt{arr}}]\rangle}} \text{(Normal-}\kappa\text{)}$$

$$\frac{\tau_{\mathtt{arr}} < \tau_{\mathtt{done}}}{\kappa_{\langle q_{\mathtt{curr}}, \tau_{\mathtt{done}}, \_\rangle} \xrightarrow{\langle \sigma, \tau_{\mathtt{arr}}\rangle}_{tran} q_{\mathtt{err}}} \text{(Error-}\kappa\text{ Tran)} \qquad \frac{\exists \langle \sigma', \tau_{\mathtt{dist}}\rangle \in \theta(q_{\mathtt{curr}}, \sigma), \tau_{\mathtt{arr}} < \Theta[\sigma'] + \tau_{\mathtt{dist}}}{\kappa_{\langle q_{\mathtt{curr}}, \tau_{\mathtt{done}}, \Theta\rangle} \xrightarrow{\langle \sigma, \tau_{\mathtt{arr}}\rangle}_{tran} q_{\mathtt{err}}} \text{(Error-}\kappa\text{ Cons)}$$

Dispatch Relation $\boxed{\rightarrow_{disp} \subseteq \mathrm{K} \times (\mathcal{P}(\Sigma) \times \mathbb{R}) \times \mathrm{K} \times (\mathcal{P}(\Sigma) \times \mathbb{R}) \times \mathcal{P}(\Lambda \times \mathbb{R})}$

Trace Relation $\boxed{\downarrow_{trace} \subseteq \mathrm{K} \times (\mathcal{P}(\Sigma) \times \mathbb{R})^* \times (\mathcal{P}(\Lambda \times \mathbb{R}))^*}$

$$\frac{\sigma \in \underset{\sigma' \in \vec{\sigma}}{argmin}(\pi_2(\delta(q_{\mathtt{curr}}, \sigma'))) \quad outs = \mu(q_{\mathtt{curr}}, \sigma) \quad \kappa_{\langle q_{\mathtt{curr}}, \_, \_\rangle} \xrightarrow{\langle \sigma, \tau_{\mathtt{arr}}\rangle}_{tran} \kappa_{next} \quad \vec{\sigma}_{rest} = \vec{\sigma}/\sigma}{\left\langle \kappa_{\langle q_{\mathtt{curr}}, \_, \_\rangle}, \langle \vec{\sigma}, \tau_{\mathtt{arr}}\rangle \right\rangle \rightarrow_{disp} \left\langle \kappa_{next}, \langle \vec{\sigma}_{rest}, \tau_{\mathtt{arr}}\rangle \right\rangle | outs}$$

$$\frac{\left\langle \kappa, \langle \vec{\sigma}, \tau_{\mathtt{arr}}\rangle \right\rangle \rightarrow_{disp} \left\langle \kappa', xs \right\rangle | outs \quad \left\langle \kappa', xs \right\rangle \downarrow_{trace} \left\langle \kappa'', outs' \right\rangle \quad outs'' = outs + outs'}{\left\langle \kappa, \langle \vec{\sigma}, \tau_{\mathtt{arr}}\rangle \right\rangle \downarrow_{trace} \left\langle \kappa'', outs'' \right\rangle} \text{(Trc-Cont)}$$

$$\frac{}{\left\langle \kappa, \langle \emptyset, \_\rangle \right\rangle \downarrow_{trace} \left\langle \kappa, \emptyset \right\rangle} \text{(Trc-Done)}$$

Network Relation $\boxed{\rightarrow_{net} \subseteq \mathcal{P}(\mathrm{K}) \times (\Sigma \times \mathbb{R})^* \times \mathcal{P}(\mathrm{K})(\Sigma \times \mathbb{R})^*}$

$$\frac{\left\langle \langle \vec{\sigma}, \tau_{\mathtt{arr}}\rangle_M, ps' \right\rangle = \mathtt{getSimPulses}(ps) \quad \kappa_M \in \vec{\kappa} \quad \langle \kappa_M, \langle \vec{\sigma}, \tau_{\mathtt{arr}}\rangle_M\rangle \downarrow_{trace} \langle \kappa'_M, outs\rangle \quad \vec{\kappa}' = \vec{\kappa}[\kappa'_M/\kappa_M] \quad ps'' = ps' + outs}{\langle \vec{\kappa}, ps\rangle \rightarrow_{net} \langle \vec{\kappa}', ps''\rangle} \text{(Net-Cont)}$$

$$\frac{\forall \langle \sigma, \tau_{\mathtt{arr}}\rangle \in ps.\ \sigma \in C.\Lambda}{\langle \_, ps\rangle \rightarrow_{net} \langle \_, ps\rangle} \text{(Net-Done)}$$

Figure 6. Semantics of the Transition, Dispatch, and Trace relation of the PyLSE Machine $\langle Q, q_0, \Sigma, \Lambda, \delta, \mu\rangle$ as well as the Network relation for larger composite designs. $\pi_i(\langle ..., x_i, ...\rangle) = x_i$ is standard tuple projection. $\Theta[\sigma \mapsto \tau]$ produces an updated mapping where σ now maps to τ. We use $S[y/x]$ to denote y replacing x in S. The helper function `getSimPulses` extracts the pulse heap ps into the earliest set of simultaneous pulses destined for the same PyLSE Machine and the rest for later use. If both x and y are heaps of pulses, we use $x + y$ to denote merging them into a single ordered heap.

Nondeterminism occurs when there are multiple simultaneous pending pulses on the heap ps going to different PyLSE Machines; the helper function `getSimPulses` chooses one before proceeding with the next.

4 PyLSE Language Design

We use the above PyLSE Machine formalism to develop a practical embedded DSL that eases the description and analysis of SCE designs at multiple levels.[3] Its abstract syntax is found in Figure 7. By being embedded in Python, we lower the barrier of entry for new users and gain the productivity benefits of using Python's libraries.

4.1 Design Levels

Cell Definition Level: Given that there is still no dominant logic scheme for SCE designs, the ability to easily define new cells is crucial for the advancement of the field. We enable this by providing a `Transitional` Python abstract class. Each SCE cell is modeled as a class that implements `Transitional`, defining the set of input and output names and a list of transitions as class attributes. Each transition in this list is represented as a Python dictionary, storing key-value pairs matching the information found in Figure 4.

[3]The language implementation is available at https://github.com/UCSBarchlab/PyLSE.

$$pt \in Port \quad st \in State \quad \sigma \in Store \quad e \in Exp \quad n \in \mathbb{Z} \quad \tau \in Time$$

$$\begin{aligned}
p \in Program &::= \textbf{ins } pt^+ \textbf{ outs } pt^+ \textbf{ cells } cell^+ \textbf{ conns } con^+ \\
cell \in Cell &::= pm \mid h \\
pm \in PyLSEMachine &::= \textbf{states } st^+ \textbf{ start } st \\
&\quad\ \ \textbf{ins } pt^+ \textbf{ outs } pt^+ \textbf{ edges } ed^+ \\
h \in Hole &::= \textbf{ins } pt^+ \textbf{ outs } pt^+ \textbf{ func } (\lambda\ pt^+\ \sigma\ \tau.e) \\
ed \in Edge &::= \textbf{priority } n \textbf{ src } st \textbf{ dest } st \textbf{ trigger } pt \\
&\quad\ \ \textbf{transtime } \tau \textbf{ firing } \mu \textbf{ constraints } \theta \\
con \in Connection &::= m.pt \leftarrow m.pt \\
m \in Entity &::= cell \mid p \\
\mu, \theta \in TimingMap &::= [pt := \tau]^*
\end{aligned}$$

Figure 7. The Abstract Syntax for the PyLSE language. A program is a collection of input and output ports, cells, and connections between them.

```
class AND(SFQ):
  _setup_time, _hold_time = 2.8, 3.0
  name = 'AND'
  inputs, outputs = ['a', 'b', 'clk'], ['q']
  transitions = [
    {'src': 'idle', 'trigger': 'clk', 'dst': 'idle',
     'transition_time': _hold_time,
     'past_constraints': {'*': _setup_time}},
    {'src': 'idle', 'trigger': 'a', 'dst': 'a_arr'},
    {'src': 'idle', 'trigger': 'b', 'dst': 'b_arr'},
    {'src': 'a_arr', 'trigger': 'b', 'dst': 'ab_arr'},
    ...
    {'src': 'ab_arr', 'trigger': 'clk', 'dst': 'idle',
     'transition_time': _hold_time,  'firing': 'q',
     'past_constraints': {'*': _setup_time}},
    {'src': 'ab_arr', 'trigger': ['a', 'b'],
     'dst': 'ab_arr'},
  ]
  jjs, firing_delay = 11, 9.2
```

Figure 8. Synchronous And Element PyLSE code. Transitions have been omitted in the space marked ....

To better understand the structure of PyLSE, we revisit the Synchronous And Element gate, originally described in Figure 5 and analyzed in Section 3.1. The PyLSE code for this cell is provided in Figure 8. It implements an abstract class `SFQ`, which is a child of the `Transitional` class mentioned previously. The `SFQ` class's purpose is to require additional attributes specific to SFQ cell design from its implementing classes. In particular, it requires that the `jjs` (the number of Josephson junctions) and `firing_delay` attributes exist on the class. `jjs` is an area metric based on the number of switching elements used by the design.

The priorities of transitions can be given explicitly, via the `priority` key in each transition dictionary, or implied by the order in which they are listed. For example, in the case of the Synchronous And Element, the transition leaving `idle` on `clk` is given before the transition leaving `idle` on `a`. Thus, the former's trigger has priority over the latter's. This priority order is isolated to transitions with the same `source` state. For example, the first and fourth transitions have different `source` states (`idle` and `a_arrived`, respectively), and thus their relative order in this list of transitions is irrelevant.

PyLSE contains a library of all the basic SCE cells [4] and provides templates for the creation of custom ones.

Hole Description Level: To facilitate the rapid prototyping and exploration of more complicated designs without the need to describe every single block via interacting transition systems, PyLSE provides the Hole Description Level. At this level, pure Python code is wrapped in a specialized interface (by implementing a `Functional` abstract class), allowing non-transition-based abstract "holes" to communicate via pulses with the rest of the system. The `Functional` class takes as initialization parameters (1) a `Callable` function mapping time-tagged input pulses to output pulses, (2) the list of input and output names, and (3) the firing delay for each output. The user can also simply wrap a Python function (with the appropriate signature) using the `hole` decorator. Note that these holes do not abide by the formal semantics of Section 3.

```
mem = defaultdict(lambda: 0)
raddr = waddr = wenable = data = 0

@pylse.hole(delay=5.0, inputs=['ra3', 'ra2', 'ra1',
↪  'ra0', 'wa3', 'wa2', 'wa1', 'wa0', 'd1', 'd0',
↪  'we', 'clk'], outputs=['q1', 'q0'])
def memory(ra3, ra2, ra1, ra0, wa3, wa2, wa1, wa0,
           d1, d0, we, clk, time):
  nonlocal raddr, waddr, wenable, data
  raddr |= ra3*8 + ra2*4 + ra1*2 + ra0
  waddr |= wa3*8 + wa2*4 + wa1*2 + wa0
  data |= d1*2 + d0
  wenable |= we
  if clk:
    if wenable:
      mem[waddr] = data
    value = mem[raddr]
    raddr = waddr = wenable = data = 0
  else:
    value = 0
  return ((value >> 1) & 1), value & 1
```

Figure 9. An example `Functional` ("hole") element modeling a memory with 16 addresses, each storing 2 bits.

An example functional element is found in Figure 9, which shows how to create a memory by wrapping a Python dictionary in a function with a pulse-communicating interface.

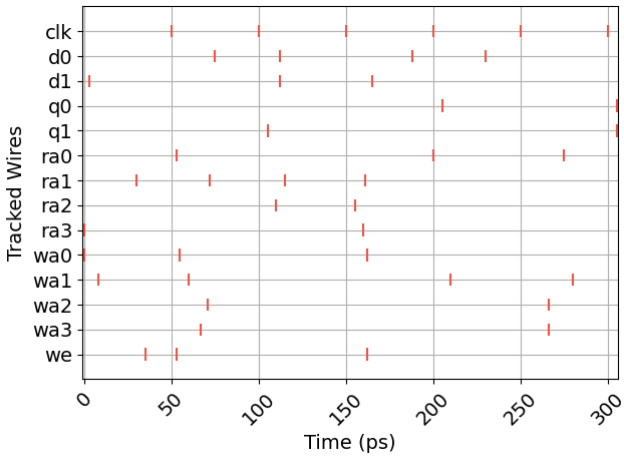
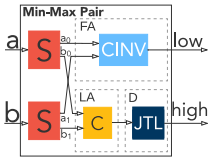


Figure 10. Graphical results of simulating the memory `Functional` class.



(a) Block diagram.

```
from pylse import s, c, c_inv, jtl
def min_max(a, b):
  a0, a1 = s(a)
  b0, b1 = s(b)
  low = c_inv(a0, b0)
  high = c(a1, b1)
  high = jtl(high, firing_delay=2.0)
  return low, high
```

(b) PyLSE code.

Figure 11. Min-max pair. Inputs a and b are duplicated by the splitters. a_0 and b_0 enter the Inverted C Element, which propagates an output pulse on low some delay after the first of its inputs arrive. a_1 and b_1 are fed into the C Element, whose output is delayed via a Josephson transmission line (for path balancing) before being emitted as the high output. The 2.0 JTL delay has been calculated based on the difference in delays between the paths to low and to high, assuming a splitter delay of 11, C Element delay of 12, and Inverted C Element delay of 14. Thus, given low, high = comp(a, b), the earlier input pulse propagates to low after 11 + 14 = 25 ps and the latter to high (likewise after 11 + 12 + 2 = 25 ps).

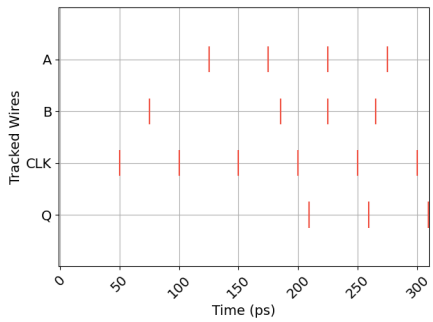
This function, memory(), takes in twelve boolean-valued arguments and a thirteenth argument, time, which is implicitly passed as the last argument to all functional elements. The read and write addresses, ra* and wa*, are split into four 1-bit inputs. A pair of nonlocal variables raddr and waddr are used to remember which address bits have been seen since the last clock pulse. If write is enabled when a clock pulse arrives, the memory is updated, the newly read value is produced as tuple of 1-bit values, and raddr and waddr are reset, ready for the next period. The arguments are internally connected to PyLSE Wires in the network. The framework automatically converts the presence of a pulse on one or more of these wires at a particular instant as a call to memory(), passing a value of 1 for each of the corresponding arguments, and the current time. Figure 10 shows the result of simulating the memory against a variety of inputs.

Full-Circuit Design Level: Nodes of Transitional and Functional class instances are interconnected with Wires and added to the circuit workspace at the Full-Circuit Design level. The code in Figure 11b provides an example of a Min-Max pair implemented with two splitters, a C Element, an Inverted C Element, and a JTL [4] following recently introduced temporal conventions [52]. Calling the function min_max(a,b) causes its constituent cells and connective wires to be instantiated via the calls to the encapsulating functions s, c, c_inv, and jtl. These functions take in wire objects and return one or more output wire objects as result. This encapsulation enables basic cells to resemble Python

[4] At a very high level, a JTL is a basic cell used for connecting other cells over larger distances, in turn adding delay to a design.

```
from pylse import inp_at, inp, and_s, Simulation
a = inp_at(125, 175, 225, 275, name='A')
b = inp_at(75, 185, 225, 265, name='B')
clk = inp(start=50, period=50, n=6, name='CLK')
out = and_s(a, b, clk, name='Q')
sim = Simulation()
events = sim.simulate()
assert events['Q'] == [209.2, 259.2, 309.2]
sim.plot()
```

(a) Exhaustively simulating a Synchronous And Element.



(b) Simulation result in graphical form.

Figure 12. Simulation of the Synchronous And Element. Pulses occur on wires A at 125.0, 175.0, 225.0, and 275.0; B at 75.0, 185.0, 225.0, and 265.0; and CLK every 50.0 ps. We check for expected pulses on Q 9.2 ps (the firing delay) after a clock period in which both A and B were received.

operators and improve language usability by updating the circuit workspace automatically. These functions also take in optional arguments, making it easy to *override* properties like firing delay, transition time for arbitrary transitions, and the number of Josephson junctions used in a particular element instance. At this level, full application implementations can be realized through the technique of elaboration-through-execution [3, 9, 13], although here, unlike traditional HDLs, the underlying primitives used by higher level generators are inherently stateful and pulse-based.

4.2 Syntactic and Semantic Checks

PyLSE provides several syntactic and semantic checks to alert the user if a design is ill-formed. At the Cell Definition level, PyLSE ensures that the list of a cell's transitions constitute a well-formed transition system. This includes ensuring the use of recognized field names, references to valid input triggers and output signal names, and the inclusion of an idle starting state. More advanced checks include the complete specification of transitions for every possible trigger and that an output occurs on at least one transition.

At the Circuit Design level, we currently check that all circuit outputs have a "fanout" of one. In SCE, the outputs of arbitrary cells cannot immediately be shared by multiple

Table 1. Functions used in the code in Figure 12a. The first four return a named wire, while `simulate()` returns a mapping from each named wire to the ordered list of pulses that appeared on it. The last two are methods on the `Simulation` class.

Function	Description
`inp_at(*times, name=None)`	Produce pulses at each time in `*times`.
`inp(start=0, period=0, n=1, name=None)`	Produce pulses starting at `start`, occurring `n` more times every `period` picoseconds.
`split(wire, n=2, names=None, **overrides)`	Split a wire `n` ways, creating `n-1` splitter elements in a binary tree.
`inspect(wire, name)`	Give a `wire` a `name` for observation during simulation.
`simulate(self, until, variability=False)`	Simulate the circuit `until` a certain time or all pulses are processed.
`plot(self)`	Produce a graph plotting the pulses against time.

inputs; instead, a *splitter* cell must be used, which is specifically designed to forward an incoming pulse to two different outgoing wires. The example in Figure 11b includes two splitter cells (lines 3 and 4) to allow `a` and `b` to be used in two different places; PyLSE reports an error on instantiation if, for example, input `a` is used in both lines 5 and 6.

4.3 Simulation

PyLSE's built-in simulator can be used to validate designs for a given set of input signals. Its design follows the principles of other discrete-event simulation frameworks [32]. So, according to the semantics provided in Figure 6, it maintains a priority heap of pending pulses tagged with their destination cells. These pulses are extracted from the heap one at a time and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and the process continues iteratively until the heap is empty or the user-defined target time is reached. This target time is needed when there are loops in the system.

```
b = inp_at(99, 185, 225, 265, name='B')
...
events = sim.simulate()
pylse.pylse_exceptions.PylseError: Error while sending
input(s) 'clk' to the node with output wire '_0':
Prior input violation on FSM 'AND'. A constraint on
transition '7', triggered at time 100.0, given via the
'past_constraints' field says it is an error to trigger
this transition if input 'b' was seen as recently as
2.8 time units ago. It was last seen at 99.0, which is
1.7999999999999998 time units to soon.
```

Figure 13. Changing the first time at which a pulse is produced on `B` in the simulation of Figure 12a rightfully results in a past constraint error due to the setup time.

Figure 12a shows how a single Synchronous And Element gets instantiated and simulated, using the functions described in Table 1. In lines 2 and 3, we create two inputs named `A` and `B`, producing four pulses on each. Line 4 creates a periodic clock signal, while lines 6 and 7 create and start a simulation object. Line 8 verifies the correctness of pulses appearing on output `Q`; here, the first appears at 209.2 ps, exactly `firing_delay` after the input pulse on `CLK` that ended the first clock period in which both `A` and `B` appeared. Line 9 produces the graph in Figure 12b. Finally, Figure 13 shows the PyLSE simulator catching a past constraints violation (the setup time constraint). The first pulse produced on `B` arrives too soon before the next pulse that arrives on `CLK`.

4.4 Correspondence with Timed Automata

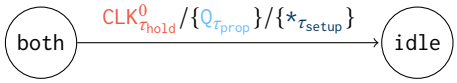
Timed Automata (TA) are a related formalism with a rich theoretical foundation, used extensively to model real-time systems with timing constraints. A Timed Automaton [2] is a finite state machine whose state transitions are guarded by conditions on a set of *resettable clocks*, defined as follows:

Definition 4.1 (Timed Automata)**.** A Timed Automaton $A = \langle L, l_0, \Sigma, C, E, I\rangle$ is a tuple where $(l \in)L$ is a set of locations, $l_{init} \in L$ is the initial location, $(\alpha \in)\Sigma$ is the set of actions, $(c \in)C$ is a set of clocks, $I : L \to \Phi(C)$ are clock invariants at each location, and

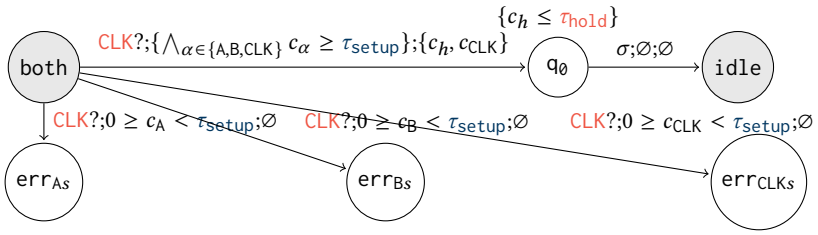
$$(e \in)E \subseteq L \times \Sigma \times \Phi(C) \times \mathcal{P}(C) \times L$$

is the set of transitions. $e = \langle l, \alpha, \varphi, \lambda, l'\rangle \in E$ is a transition from location l to l' on action α, φ is the guard specifying conditions that must be true on the clocks, and λ is the set of clocks to be reset after the transition.

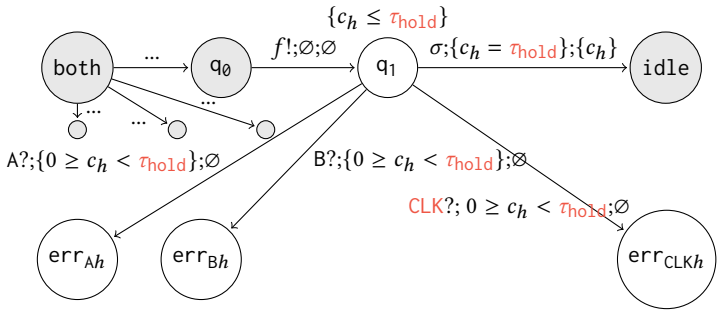
To directly obtain the benefits of TA, we convert a PyLSE Machine to a network of Timed Automata running in parallel. Figure 14 graphically shows this conversion process for a single edge of the Synchronous And Element. This is the same edge highlighted in Figure 5, but here we have replaced the state named `a and b arrived` with `both` because of space constraints. At a high level, this process works by expanding the edges from the original PyLSE Machine into TA transition sequences. We first create TA clocks for each PyLSE Machine input — c_{A}, c_{B}, and c_{CLK} — in addition to a clock, c_h, that measures the time elapsed on transitions. These clocks are available to all edges of this TA. Given edge $\mathsf{CLK}^{0}_{\tau_{\mathtt{hold}}}/\{\mathsf{Q}_{\tau_{\mathtt{prop}}}\}/\{*_{\tau_{\mathtt{setup}}}\}$ emerging from state `both`, translation proceeds incrementally. The input symbol `CLK` of the PyLSE Machine becomes a TA channel `CLK` on which messages are only received by this automaton. The time it takes to complete the transition, $\tau_{\mathtt{hold}}$, becomes part of the inequality in both location $\mathsf{q}_{\emptyset}$'s invariant and in the guard involving clock c_h as part of the final edge to `idle`. In addition, clocks



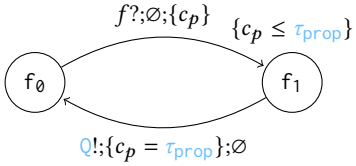
(a) Original PyLSE Machine transition, moving from `both` to `idle` on CLK only if τ_{hold} time has passed, producing output Q after τ_{prop} time. It is an error if any inputs (i.e. $*$) arrived in the last τ_{setup} time units prior to starting this transition. The priority 0 is ignored here in isolation.



(b) (Intermediary step) Expanding the original transition into two intermediate TA transitions that handle receiving a message on channel CLK (corresponding to original symbol CLK) (left edge), checking for the transition time to have passed (right edge), and erroring out (to err$_{As}$, err$_{Bs}$, or err$_{CLKs}$) if the τ_{setup} is violated.



(c) (Final step, part 1) Further expanding the transition to include transitions for firing output f and erroring out (to err$_{Ah}$, err$_{Bh}$, or err$_{CLKh}$) if unexpected inputs are received during a transitionary period.



(d) (Final step, part 2) Auxiliary TA created for modeling firing delay. The TA in Figure 14c sends a message on channel f, which is received here. After τ_{prop} time units, output finally appears on output channel Q. This firing TA, including a fresh clock c_p, is duplicated by a soaking factor $s = \lceil \tau_{prop}/\tau_{hold} \rceil$ to allow the network to fire again if needed during the transition.

Figure 14. Expanding a PyLSE Machine transition into its corresponding TA transitions, using an edge from the Synchronous And Element (for brevity, we've replaced the state named `a and b arrived` with `both`). We assume clocks c_h, c_s, and c_p and channels A, B, CLK, f and Q. Shaded states (or ... edges) indicate old states (edges) repeated from the previous figure.

c_A, c_B, and c_{CLK} are compared against the past constraint value, τ_{setup}, in the first edge's guard. The TA goes to an error state if these constraints are violated and otherwise transitions to q_0. Figure 14b is the result of this first conversion.

To detect inputs while in the transitional period, Figure 14c inserts three additional states – err_a, err_b and err_{clk} – to cover all possible input messages. Figure 14c also adds the intermediate state q_1, for sending a firing message f to an auxiliary TA created in Figure 14d and for setting up the clock that is used for checking that the transitional timing period has been satisfied before going to state `idle`. The auxiliary TA in Figure 14d is created entirely *alongside* the previous TA. When it receives a message f to fire, it waits the designated firing delay time τ_{prop} before sending a message on channel Q. Here, producing output Q in the original PyLSE Machine corresponds to sending a message on the channel Q. This channel, created solely for sending, allows an output action and transition to occur in parallel.

There is a significant increase in complexity as one moves down from the PyLSE Machine to the TA. For example, Figure 14 shows that at least 12 TA locations and 11 edges must be created to describe a *single* PyLSE Machine transition. The entire resultant TA network for a single Synchronous And Element PyLSE Machine has 102 locations and 110 edges. PyLSE properly encapsulates this complexity, allowing this much larger TA network to be represented by the four states and twelve edges of the original PyLSE Machine of Figure 5.

Timed-arc Petri nets also offer a broad, descriptive formalism for concurrent systems. However, we discovered that TA offer a better balance between expressivity and usability. Specifically, computing reachability for unbounded timed-arc Petri nets is undecidable, while for TA it is decidable in PSPACE [8]. Additionally, the complexity results for other constructions such as bisimilarity are not any more performant for Petri nets than TA, and TA are equivalent in expressive power to bounded timed-arc Petri nets [10].

5 Evaluation

The goal of our evaluation is to prove the following claims:

Claim 1. *PyLSE can be used to accurately model the functional and timing behavior of basic SCE cells and larger designs.*

Claim 2. *PyLSE offers significant productivity gains over state-of-art HDLs for designing and simulating basic SCE cells and larger designs.*

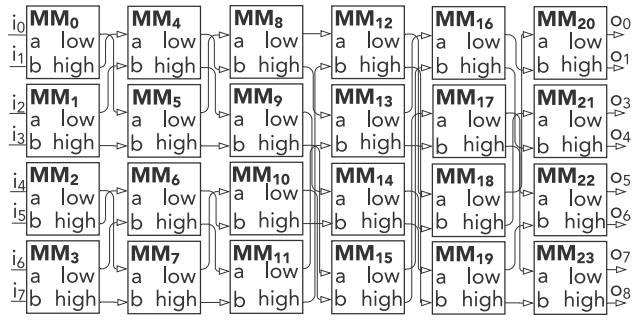


Figure 15. A eight-input bitonic sorter, composed of twenty-four comparators (see Figure 11a). It takes eight individual input wires (i_0 through i_7), which are produced in arrival time order, after some network delay, on o_0 through o_7.

Claim 3. *PyLSE can be used in conjunction with a state-of-the-art model checker to formally verify properties of basic SCE cells and larger designs.*

To evaluate these claims, we implemented 16 basic cells (constituting the PyLSE standard library) plus six larger designs as listed in Table 3. These constitute a representative set of functions showing that PyLSE transition systems can accurately capture the functional behavior of SCE cells using pulse-based signaling and systems built out of such cells. For example, we have used PyLSE to model both synchronous RSFQ designs and asynchronous xSFQ and temporal SCE designs. As far as we know, there are no open source cell libraries available that contain all the basic cells we list in Table 3, making direct comparisons difficult.

5.1 Circuit Simulation Comparison

Circuit designers perform simulations with low-level languages like SPICE [40] and WRSpice [58] to create analog gate models using fundamental electrical components. However, this process can be time-consuming and requires significant domain expertise. The PyLSE abstraction complements this process; the nominal timing values found through these detailed circuit-level simulations inform the models of the gates via their respective PyLSE Machines. It is through this abstraction that PyLSE can improve productivity by making it easier to scale and simulate larger designs before physically implementing them.

However, in the analog domain, loading effects and additional buffering stages used to improve signal fidelity can change these observed timing values when two or more gates are connected together. These in turn cause small timing differences to be observed between PyLSE and circuit simulations in the cases of larger designs. To compensate for such variations, PyLSE allows you to express the timing behavior of an SCE cell as a distribution.

Thus, a key to developing a successful simulator at a different level of abstraction is to verify that the two match

Table 2. Simulation times of PyLSE vs. schematic-level models. For the C and InvC elements, size refers to the number of transitions in the DSL (≈ the number of lines), and for the rest, the number of lines. The number of schematic lines reported is the size of the unflattened netlist.

Name	Schematic (Cadence)		PyLSE	
	Lines	Time (s)	Size	Time (s)
C	81	2.840	6	0.000298
InvC	87	2.987	6	0.000336
Min-Max Pair	140	4.608	5	0.000617
Bitonic Sort 8	250	52.565	24	0.003857

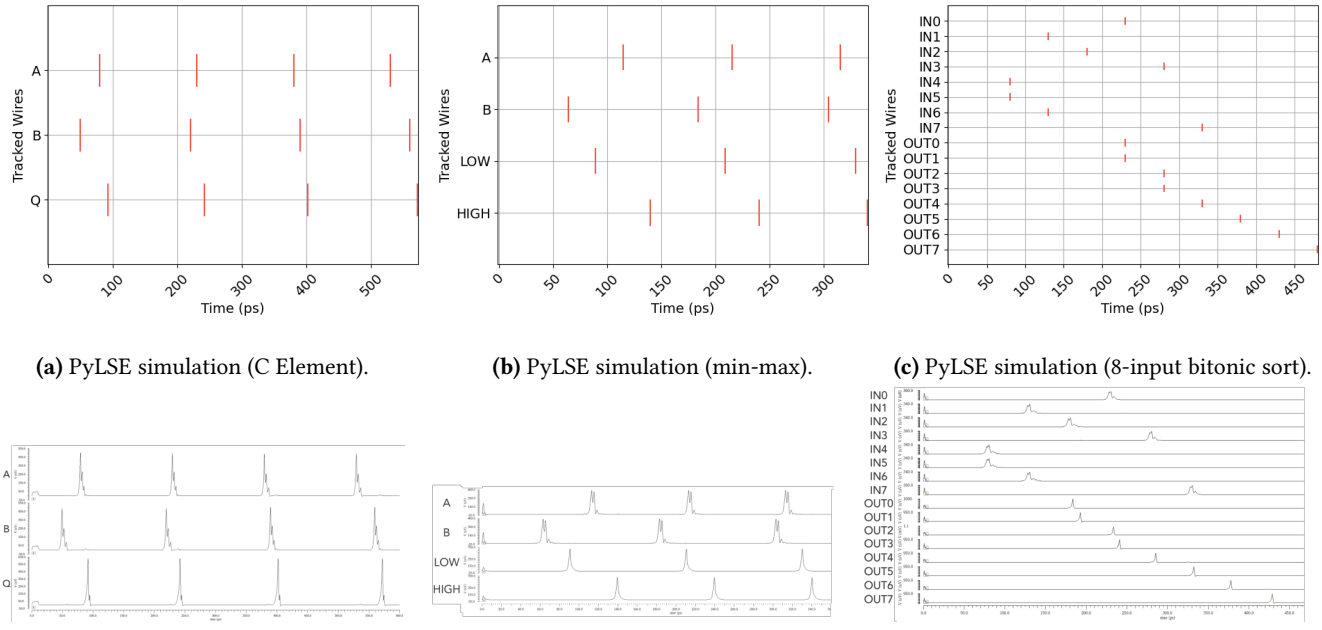
in spite of design size. In this section, we demonstrate this for PyLSE by focusing on an 8-input bitonic sorter and the cells that compose it. A bitonic sorter [5] is a parallel sorting network made up of many min-max pair blocks (Figure 11), connected like in Figure 15. To validate the accuracy of PyLSE, we compare the results generated by running the four designs shown in Table 2 against circuit-level simulations. For these circuit-level simulations, we use the Cadence Virtuoso suite and a process design kit (PDK) corresponding to the state-of-the-art MITLL SFQ5ee fabrication process. Pulses are supplied through a Direct Current-to-SFQ converter fed by a current source, while the pulses are probed directly for voltage measurement.[5]

Figure 16 compares three design simulations in PyLSE and a circuit schematic simulator[6]. Figures 16a (PyLSE) and 16d (schematic) simulate the C Element. Given identical inputs and the C cell's propagation delay (12 ps), the output times of both simulations match exactly. The PyLSE waveform of the min-max pair is show in Figure 16b, and its circuit waveform in Figure 16e (SPICE). The circuit model's propagation delay along all paths is 22 ps, while the PyLSE model's propagation delay is 25 ps. The discrepancy comes because the given PyLSE design was created as a pure composition of the individual cells. When combined together at the circuit schematic level, however, the entire system exhibits a smaller total propagation delay than what would be assumed from the sum of its parts, due to the parasitic and loading effects mentioned previously. However, the delay of each individual cell in PyLSE can be tuned, with optional variability added (see Section 5.2), to match the circuit schematic behavior more closely, if desired.

The PyLSE waveform of the 8-input bitonic sorter is displayed in Figure 16c and its circuit waveform in Figure 16f. The composability issue creeps up here as well: the circuit model's entire propagation delay is between 100 and 110 ps, while a purely compositional delay would equal the min-max circuit model's delay (22 ps) multiplied by the depth (6) of the

[5]In practice, the readout is enabled by SFQ-to-DC [36] converters because direct wire probing is not feasible.

[6]The Inverted C Element waveforms have been omitted for space.



(a) PyLSE simulation (C Element). **(b)** PyLSE simulation (min-max). **(c)** PyLSE simulation (8-input bitonic sort).

(d) Circuit simulation (C Element). **(e)** Circuit simulation (min-max). **(f)** Circuit simulation (8-input bitonic sort).

Figure 16. PyLSE vs circuit simulation results for the C Element, min-max pair and 8-input bitonic sorter.

network, i.e. $22 * 6 = 132$ ps. The PyLSE design, with a total delay of $25 * 6 = 150$, nevertheless functions correctly. For example, the pulse arriving on input IN4 (the earliest input pulse) is produced 150 ps later on OUT0, and more generally, the output pulses appear in rank order as expected. Table 2 compares sizes and simulation times of these designs. The PyLSE versions are an average of 16.6× *smaller* than their circuit schematic counterparts and take an average 9879× less time to simulate. These example simulations demonstrate an important trade-off: the extremely high accuracy of the analog design level versus the scalability and rapid prototyping of PyLSE.

5.2 Simulation and Dynamic Correctness Checks

In this subsection, we harness the rich features of Python to quickly validate our designs for correctness. In particular, we can use the `events` dictionary that PyLSE returns from a simulation run (see Figure 12a) to assert various correctness properties. Several examples follow; similar tests were performed for all 22 designs shown in Table 3. More details are found in our artifact and GitHub repository.

2x2 Join. The 2x2 Join element is a dual-rail logic primitive that takes in two pairs of inputs that are logical complements — in this case, A_T and A_F, and B_T and B_F — and produces one of four outputs depending on which pair of inputs have arrived. For its complete PyLSE specification, 12 transitions are needed. A requirement for this cell to function correctly is to interleave a B_T or B_F pulse between subsequent A_T and A_F pulses and vice versa. This can be written succinctly as follows:

```
inputs = sorted(((w, p) for w, evs in events.items()
  for p in evs if w in ('A_T', 'A_F', 'B_T', 'B_F')),
  key=lambda x: x[1])
zipped = list(zip(inputs[0::2], inputs[1::2]))
assert all(x[0] != y[0] for x, y in zipped)
```

Race Tree. A race tree [51] is a decision tree that uses the principles of race logic to return a winner label based on a set of internal decision branches. We implement a race tree in PyLSE by composing 18 basic SFQ cells together in a total of 20 lines of code. A fundamental correctness property of these trees is that they return only one output label for each set of input pulses. The following assertion encodes this condition using the `events` dictionary of before:

```
assert sum(len(l) for o, l in events.items()
           if o in ('a', 'b', 'c', 'd')) == 1
```

8-input Bitonic Sorter. A bitonic sorter is correct if, given a single pulse on each input at an arbitrary time (spaced far enough apart to satisfy transition time constraints), the outputs appear in rank order. This property can be expressed as follows, assuming the first output that should appear is named O0, followed by O1, etc. until the last output ON for some power-of-two N:

```
out_events = {e for e in events.items()
              if e[0].startswith('o')}
ordered_names = sorted(out_events.keys())
ranked = [es for _, es in sorted(out_events.items(),
          key=lambda x: ordered_names.index(x[0]))]
assert all(len(es) == 1 for es in ranked)
```

```
assert all(x[0] <= y[0] for x, y
           in zip(ranked, ranked[1:]))
```

Evaluating Robustness Given Timing Variability. In the circuit world, propagation delays of these basic cells vary from the expected values when chaining them together. This is apparent in the bitonic sort example of Section 5.1, where the circuit's delay varied between 100 and 110 ps. Such variance can lead to pulses arriving at their destination cells too early or late, causing the design to fail unexpectedly. At a PyLSE Machine level, these failures are detected by violations of transition and past constraint times during simulation or by erroneous outputs seen after simulation, and might signify that the network needs to be redesigned to make it less sensitive to variability. PyLSE makes it easy to add variability to existing designs and evaluate their robustness in the presence of these variations; simply pass the flag `variability=True` to `simulate()`. Every individual propagation delay that occurs during the simulation will then have a small amount of delay, by default taken from a Gaussian distribution, added to or subtracted from it. The `variability` argument can be used to specify the cell types or the individual cell instances where the default variability should be added, or it can be set to a user-defined function for even greater fine-tuning.

5.3 Model Checking in UPPAAL

Model checking [12] is a formal verification technique used to check that a particular property, typically written in a temporal logic, holds for certain states on a given model of a system. Before it can be used, however, a *model* of the system must be created. Timed Automata is one such model, and as we have shown in Section 4, PyLSE can automatically transform PyLSE Machines into a network of communicating Timed Automata; in this way, designs written in PyLSE *are* the models themselves, and immediately amenable to formal verification.

We have chosen to integrate with UPPAAL, a state-of-the-art framework for modeling real-time systems based on TA [7]. The conversion process is straightforward: the PyLSE circuit is traversed, with every transition of every element being converted according to the steps in Figure 14 into a network UPPAAL-flavored TA. The result is saved to an XML file, which can then be simulated in UPPAAL or verified against certain properties on the command line via the `verifyta` program their distribution provides.

Query 1: Correctness. To verify that our translation process works, we automatically converted all 16 basic cells and six larger designs into UPPAAL, as shown in Table 3, where we note the resulting size of the TA network. Once in UPPAAL, we checked that their internal simulator agrees with ours from an input/output perspective. We also automatically generate a correctness formula in UPPAAL-flavored timed computation tree logic (TCTL) [6, 21] for each, based on a given PyLSE simulation's events, to formally verify that the given design generates the expected output. For example, here is a PyLSE-generated TCTL formula for the correctness of min-max pair, given pulses on A at 115, 215, and 315, on B at 64, 184, and 304, and a network delay of 25 ps:

```
A[] (((firingauto3.fta_end imply ((global == 890) ||
       (global == 2090) || (global == 3290))) &&
      (firingauto4.fta_end imply ((global == 890) ||
       (global == 2090) || (global == 3290))) &&
      (firingauto5.fta_end imply ((global == 890) ||
       (global == 2090) || (global == 3290)))) &&
      ((firingauto12.fta_end imply ((global == 1400) ||
       (global == 2400) || (global == 3400)))))
```

At the top of this formula, *A* is a path quantifier that expresses "for all subsequent time points", while [] is a branch quantifier meaning "for all possible branches." The `firingauto*` correspond to firing TA instances, and `fta_end` is the location in that instance that immediately follows sending a fire message to a particular network output sink. As many firing TA may be associated with each network output (see Figure 14d), there are multiple states to check for each time. This says that it is only possible to produce a pulse at the given output at the given time. These times have been upscaled to integers to meet the requirements UPPAAL places on numbers involved in clock constraints; thus `global == 2090` is 209.0 ps in PyLSE.

In Table 3, we also show the time it took to verify this property (customized to each cell). For the basic cells and the min-max pair, verification consistently took less than 1 second. The race tree, with 440 locations, took 127 seconds and explored 262559 states, while the synchronous full adder, with nearly 43% more locations, took 669 seconds (5.26×) and visited 7.077× more states. Model checking becomes infeasible due to the state explosion as we reach the bitonic sorters and xSFQ [54] full adder, which failed to finish in a day. Table 3 also shows how much larger the network of TA is compared to the original PyLSE Machines. On average, each cell (i.e. PyLSE Machine) requires 3.02 UPPAAL TA, each PyLSE Machine state requires 18.99 UPPAAL locations, and each PyLSE Machine transition requires 9.05 UPPAAL transitions.

Query 2: Unreachable Error States. Our translation process inserts error states that are entered when transition time or past constraint violations occur (for example, $\texttt{err}_{Ah}$ and $\texttt{err}_{As}$, respectively, from Figure 14). Since these states have no outgoing edges, they cannot respond to additional input nor allow time to pass and so are *terminal*. Entering such a state would deadlock the TA, and verifying that no deadlock occurs (i.e. `A[] not deadlock`) would normally be sufficient to show that the inputs to a design meet timing constraints. Unfortunately, this form of deadlock detection is not useful for our purposes, since "good" deadlock also occurs when the sequence of user-defined inputs has been exhausted and no more cells can progress. Instead, we automatically generate an UPPAAL verification query that checks that it is

Table 3. Basic cells (first 16 rows) and larger designs (last six rows) implemented in PyLSE. Each has been validated via PyLSE simulation for functional correctness and timing constraint violation detection, and automatically converted into TA that have been simulated and verified in UPPAAL. The PyLSE columns display counts for size, cells, states, and transitions; for basic cells, these are numbers for an individual cell, while for the larger designs, it is the accumulation of every instantiated cell in the network. The size corresponds to the number of transitions written in the DSL (roughly equal to the number of lines) for basic cells, and the number of lines for the larger designs. The first four UPPAAL columns are the number of TA, locations, transitions, and channels in the cell's generated TA network, while the latter two columns contain the time to verify the Queries 1 and 2 listed in Section 5.3 and the number of total states explored (only one number is listed in each column if the results for Queries 1 and 2 were the same). It took less than 1 second to simulate all of these designs in PyLSE.

Name	PyLSE				UPPAAL						Comparison		
	Size	Cells	States	Tran.	TA	Locs.	Tran.	Chan.	Time (s)	States	TA/Cells	Locs./States	Tran.(U)/Tran.(P)
C	6	1	3	6	2	39	42	3	<1	38	2	13	7
InvC	6	1	3	6	4	45	48	3	<1	69	4	15	8
M	2	1	1	2	2	17	18	3	<1	37	2	17	9
S	1	1	1	1	3	13	13	3	<1	56	3	13	13
JTL	1	1	1	1	2	9	9	2	<1	17	2	9	9
And	11	1	4	12	5	102	110	4	<1	69	5	25.5	9.17
Or	4	1	2	6	2	49	53	4	<1	48	5	24.5	8.83
Nand	12	1	4	12	2	95	103	4	<1	42	2	23.75	8.58
Nor	6	1	2	6	2	49	53	4	<1	36	2	24.5	8.83
Xor	9	1	3	9	3	75	81	4	<1	45	3	25	9
Xnor	12	1	4	12	2	94	102	4	<1	45	2	23.5	8.5
Inv	4	1	2	4	3	30	32	3	<1	14	3	15	8
DRO	4	1	2	4	2	27	29	3	<1	11	2	13.5	7.25
DRO SR	6	1	2	6	2	49	53	4	<1	23	2	24.5	8.83
DRO C	4	1	2	4	3	31	33	4	<1	14	3	15.5	8.25
2x2 Join	20	1	5	20	5	206	221	8	<1	58	5	41.2	11.05
Min-Max	5	5	9	15	24	149	155	14	<1	2471	4.8	16.56	10.33
Race Tree	16	18	32	56	50	440	464	54	127/84	262559	2.78	13.75	8.29
Adder (Sync)	13	19	33	71	57	627	665	62	669/515	1858153	3	19	9.37
Adder (xSFQ)	31	83	121	183	193	1449	1511	211	∞	N/A	2.33	11.98	8.26
Bitonic Sort 4	6	30	54	90	144	894	930	84	∞	N/A	4.8	16.56	10.33
Bitonic Sort 8	24	120	216	360	576	3576	3720	336	∞	N/A	4.8	16.56	10.33

impossible to reach any error state in the network (here, for the min-max pair):

```
A[] not (c0.C_err_a_1 || c0.C_err_a_11 || c0.C_err_a_16 ||
         ...18 more lines...
         c_inv0.C_INV_err_b_8 || c_inv0.C_INV_err_b_9 ||
         s0.S_err_a_1 || s0.S_err_a_2 || jtl0.JTL_err_a_1 ||
         jtl0.JTL_err_a_2 || s1.S_err_a_1 || s1.S_err_a_2)
```

UPPAAL explores the same number of states as for Query 1 in under one second for all basic cells, with the larger designs similarly encountering exponential blowup difficulties. If the above property is not satisfied, UPPAAL will return a trace showing the path that led to the particular error state.

As of this writing, additional properties must be explicitly written out in UPPAAL's DSL for expressing TCTL formulas. As far as we know, we are the first to use *timed automata-based model checking* to check the correctness of SFQ circuits.

6 Related Work

Existing HDLs. Existing HDLs, like Verilog [56], model SCE timing constraints by coupling asynchronously-updated registers with complicated series of conditionals to track whether these constraints are satisfied [1, 28, 33, 60]. Designs using this approach have many downsides:

- They tend to be extremely verbose, spanning tens to hundreds of lines per cell module. For example, in [18], 90 lines of codes were needed to model a destructive readout (DRO) cell, while the PyLSE Machine equivalent takes four lines. Similarly, a model of the OR cell in [37] takes 18 lines of Verilog, with an autogenerated model taking 58 lines of Verilog in [44].
- A number of ambiguous internal signals must be generated for synchronization purposes. For example, for the implementation of said DRO cell, five edge-triggered `always` blocks and three artificial synchronization signals were required.
- There are no clear boundaries between functional and timing specification, leading to obfuscated code and an enlarged surface for programming bugs.
- They rely on the peculiar semantics of Verilog or the chosen simulator, instead of being based on a suitable formal foundation.

Recent approaches [49, 50] are more modular and compact, but the resemblance of their proposed coding scheme to

multithreaded socket programming raises the barrier to entry and again makes them prone to bugs. Finally, other approaches [17, 38, 39, 44] automatically extract state machine models and timing characteristics of SFQ cells from SPICE files, but in the end, still use them to generate Verilog HDL code that must be integrated with the rest of the user-coded design. PyLSE can serve as a more compact and easier-to-comprehend way to analyze such models, but more importantly it has well-defined methods by which such state models compose into larger circuits – something not found in any of that work. Besides this, PyLSE allows for the easy modeling of proprietary or experimental SCE cells, where only their Mealy machine description is publicly available (and not their schematic implementation).

Existing Simulators and Logic Synthesis Tools. SFQ cell functionality is commonly verified through analog simulators catered to the unique physics of superconducting devices [15, 16, 48, 57]. Other tools such as PSCAN and PSCAN2 perform similar timing analyses, and are geared towards optimizing circuit-level parameters based on device switching events internal to the cells [42, 43, 46]. By lifting the focus to a higher level of abstraction in PyLSE, an implementation gap emerges between these simulators and PyLSE machines. While there are no theoretical limitations that prohibit the translation of PyLSE machines to schematic models, we consider such hardware synthesis to be a separate problem outside the scope of this paper. We foresee the integration of PyLSE with SCE-oriented EDA tools, such as IARPA's SuperTools, upon their public release.

Functional and Dataflow Languages. There have been efforts in the past to describe traditional hardware using dataflow programming languages. The language Lustre, a modeling framework for reactive, real-time systems, has been used for deriving an automaton from code and subsequently model checking it for safety properties[20]. The language Esterel has similarly been used to describe hardware that is then translated into equation systems inside the theorem prover HOL, motivating the possibility of formal analysis of circuit correctness as well as circuit synthesis [45]. PyLSE differs in a few respects. While dataflow languages describe hardware as a set of recursive equations, PyLSE offers a straightforward way to describe arbitrary SCE cells as transition systems, which matches the intuitions of the SCE community. Further, work using dataflow languages has focused on synchronous programs which orchestrate events and data flow according to one or more clocks. Meanwhile, PyLSE makes no requirements on synchrony, allowing the designer to more easily describe circuits with or without clocks.

Verification. There have been many attempts to formally check the correctness of SCE designs at the HDL level. Recent work [29] uses a delay-based time frame model, which assumes that pulses arrive periodically according to a unique clock period. This assumption allows them to discretize the behavior of these pulse-based systems into a verifiable synchronous model. PyLSE instead imposes no requirements about clock periodicity and therefore is also able to model systems that include asynchronous cells. VeriSFQ [59] is a semi-formal verification framework that uses UVM [55] to validate that designs are properly path-balanced, have correct fanout, and that all synchronous gates receive a clock signal. In comparison, PyLSE is an entirely new DSL for SCE design, statically preventing the creation of designs with these basic issues, and so a formal framework for checking them is unneeded. Finally, qMC [37] relies on SMT-based model checkers to verify the correct functionality of post-synthesis netlists via SystemVerilog assertions. However, their gate models do not include information on hold or setup times or propagation delay, such that outputs take a single time unit to go high. PyLSE instead represents and model checks against these timing constraints via a Timed Automata-based model checker like UPPAAL.

7 Conclusion

In this paper, we presented PyLSE, a language for the design and simulation of pulse-based systems like superconductor electronics (SCE). PyLSE simplifies the process of precisely defining the functional and timing behavior of SCE cells using a new transition-system based abstraction, which we call the PyLSE Machine. It facilitates a multi-level design approach by allowing the construction of scalable SCE systems through the mix of basic transition-based cells and higher-level abstract design models. We evaluate PyLSE by simulating and dynamically checking the correctness of 22 different designs, comparing these simulations against analog SPICE models, and verifying their timing constraints using the UPPAAL model checker. Compared to analog circuit designs, PyLSE designs take 16.6× fewer lines code and take several orders of magnitude less time to simulate, all while maintaining the needed level of timing accuracy. Compared with specifications directly made using Timed Automata, PyLSE requires 18.9× fewer states and 9.0× fewer transitions. We believe, with the end of traditional transistor scaling, pulse-based logic systems will only continue to grow in importance. PyLSE, with its expressive timing, composable abstractions, and connection to well-understood theory, has the potential to provide a new foundation for that growth for years to come.

Acknowledgments

We thank our shepherd, Sara Achour, and the anonymous reviewers for their excellent suggestions on improving the paper. This material is based upon work supported by the National Science Foundation under Grants No. 1763699, 2006542, and 1717779.

References

[1] V. Adler, Chin-Hong Cheah, K. Gaj, D. K. Brock, and E. G. Friedman. 1997. A Cadence-based design environment for single flux quantum circuits. *IEEE Transactions on Applied Superconductivity* 7, 2 (1997), 3294–3297. https://doi.org/10.1109/77.622058

[2] Rajeev Alur and David L. Dill. 1994. A Theory of Timed Automata. *Theoretical Computer Science* 126, 2 (April 1994), 183–235. https://doi.org/10.1016/0304-3975(94)90010-8

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[4] R. S. Bakolo. 2011. *Design and implementation of a RSFQ superconductive digital electronics cell library.* Master's thesis. University of Stellenbosch.

[5] K. E. Batcher. 1968. Sorting Networks and Their Applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference* (Atlantic City, New Jersey) *(AFIPS '68 (Spring))*. Association for Computing Machinery, New York, NY, USA, 307–314. https://doi.org/10.1145/1468075.1468121

[6] Gerd Behrmann, Alexandre David, and Kim G. Larsen. 2004. A Tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures*, Marco Bernardo and Flavio Corradini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 200–236. https://doi.org/10.1007/978-3-540-30080-9_7

[7] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. 1996. UPPAAL — a tool suite for automatic verification of real-time systems. In *Hybrid Systems III*, Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 232–243.

[8] Béatrice Berard, Franck Cassez, Serge Haddad, Didier Lime, and Olivier Henri Roux. 2005. Comparison of the Expressiveness of Timed Automata and Time Petri Nets. In *FORMATS 2005 - 3rd International Conference on Formal Modeling and Analysis of Timed Systems (Lecture Notes in Computer Science, Vol. 3829)*. Springer-Verlag, Uppsala, Sweden, 211–225. https://doi.org/10.1007/11603009_17

[9] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '98)*. Association for Computing Machinery, New York, NY, USA, 174–184. https://doi.org/10.1145/289423.289440

[10] Joakim Byg, Kenneth Yrke Jørgensen, and Jiří Srba. 2009. An Efficient Translation of Timed-Arc Petri Nets to Networks of Timed Automata. In *Formal Methods and Software Engineering*, Karin Breitman and Ana Cavalcanti (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 698–716.

[11] Ruizhe Cai, Ao Ren, Olivia Chen, Ning Liu, Caiwen Ding, Xuehai Qian, Jie Han, Wenhui Luo, Nobuyuki Yoshikawa, and Yanzhi Wang. 2019. A Stochastic-Computing Based Deep Learning Framework Using Adiabatic Quantum-Flux-Parametron Superconducting Technology. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 567–578. https://doi.org/10.1145/3307650.3322270

[12] Edmund M. Clarke and E. Allen Emerson. 1982. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, Dexter Kozen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–71.

[13] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. 2017. A pythonic approach for rapid hardware prototyping and instrumentation. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Ghent, Belgium, 1–7. https://doi.org/10.23919/FPL.2017.8056860

[14] Leon N. Cooper. 1956. Bound Electron Pairs in a Degenerate Fermi Gas. *Physical Review* 104, 4 (Nov. 1956), 1189–1190. https://doi.org/10.1103/PhysRev.104.1189

[15] J. A. Delport, K. Jackman, P. l. Roux, and C. J. Fourie. 2019. JoSIM—Superconductor SPICE Simulator. *IEEE Transactions on Applied Superconductivity* 29, 5 (2019), 1–5. https://doi.org/10.1109/TASC.2019.2897312

[16] E. S. Fang and T. Van Duzer. 1989. A Josephson integrated circuit simulator (JSIM) for superconductive electronics application. In *Extended Abstracts of 1989 Intl. Superconductivity Electronics Conf. (ISEC '89)* (Tokyo, Japan). 407–410.

[17] Coenrad J. Fourie. 2018. Extraction of DC-Biased SFQ Circuit Verilog Models. *IEEE Transactions on Applied Superconductivity* 28, 6 (2018), 1–11. https://doi.org/10.1109/TASC.2018.2829776

[18] Kris Gaj, Chin-Hong Cheah, E.G. Friedman, and M.J. Feldman. 1997. Functional modeling of RSFQ circuits using Verilog HDL. *IEEE Transactions on Applied Superconductivity* 7, 2 (1997), 3151–3154. https://doi.org/10.1109/77.622000

[19] Kris Gaj, Eby G. Friedman, and Marc J. Feldman. 1997. Timing of Multi-Gigahertz Rapid Single Flux Quantum Digital Circuits. In *High Performance Clock Distribution Networks*, Eby G. Friedman (Ed.). Springer US, Boston, MA, 135–164. https://doi.org/10.1007/978-1-4684-8440-3_11

[20] Nicolas Halbwachs, Daniel Pilaud, Farid Ouabdesselam, and Anne-Cecile Glory. 1989. Specifying, programming and verifying real-time systems using a synchronous declarative language. In *International Conference on Computer Aided Verification*. Springer, 213–231.

[21] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. 1992. Symbolic model checking for real-time systems. In *[1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*. IEEE, Santa Cruz, CA, USA, 394–406. https://doi.org/10.1109/LICS.1992.185551

[22] Adam Holmes, Mohammad Reza Jokar, Ghasem Pasandi, Yongshan Ding, Massoud Pedram, and Frederic T. Chong. 2020. NISQ+: Boosting quantum computing power by approximating quantum error correction. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 556–569. https://doi.org/10.1109/ISCA45697.2020.00053

[23] D. Scott Holmes, Alan M. Kadin, and Mark W. Johnson. 2015. Superconducting Computing in Large-Scale Hybrid Systems. *Computer* 48, 12 (2015), 34–42. https://doi.org/10.1109/MC.2015.375

[24] D. Scott Holmes, Andrew L. Ripple, and Marc A. Manheimer. 2013. Energy-Efficient Superconducting Computing—Power Budgets and Requirements. *IEEE Transactions on Applied Superconductivity* 23, 3 (2013), 1701610–1701610. https://doi.org/10.1109/TASC.2013.2244634

[25] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.

[26] K. Ishida, I. Byun, I. Nagaoka, K. Fukumitsu, M. Tanaka, S. Kawakami, T. Tanimoto, T. Ono, J. Kim, and K. Inoue. 2020. SuperNPU: An Extremely Fast Neural Processing Unit Using Superconducting Logic Devices. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 58–72. https://doi.org/10.1109/MICRO50266.2020.00018

[27] B.D. Josephson. 1962. Possible new effects in superconductive tunnelling. *Physics Letters* 1, 7 (1962), 251–253. https://doi.org/10.1016/0031-9163(62)91369-0

[28] Naveen Katam, Soheil Nazar Shahsavani, Ting-Ru Lin, Ghasem Pasandi, Alireza Shafaei, and Massoud Pedram. 2017. SPORT Lab SFQ Logic Circuit Benchmark Suite. https://ceng.usc.edu/techreports/2017/Pedram%20CENG-2017-1.pdf. Accessed: 2021-10-22.

[29] T. Kawaguchi, K. Takagi, and N. Takagi. 2015. A Verification Method for Single-Flux-Quantum Circuits Using Delay-Based Time Frame Model. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* 98-A (2015), 2556–2564.

[30] A. Krasniewski. 1993. Logic simulation of RSFQ circuits. *IEEE Transactions on Applied Superconductivity* 3, 1 (1993), 33–38. https://doi.org/10.1109/77.233410

[31] K.K. Likharev and V.K. Semenov. 1991. RSFQ logic/memory family: a new Josephson-junction technology for sub-terahertz-clock-frequency digital systems. *IEEE Transactions on Applied Superconductivity* 1, 1 (1991), 3–28. https://doi.org/10.1109/77.80745

[32] Norm Matloff. 2008. Introduction to discrete-event simulation and the simpy language. *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August* 2, 2009 (2008), 1–33.

[33] F. Matsuzaki, N. Yoshikawa, M. Tanaka, A. Fujimaki, and Y. Takai. 2003. A behavioral-level HDL description of SFQ logic circuits for quantitative performance analysis of large-scale SFQ digital systems. *Physica C: Superconductivity* 392-396 (2003), 1495 – 1500. https://doi.org/10.1016/S0921-4534(03)00775-5 Proceedings of the 15th International Symposium on Superconductivity (ISS 2002): Advances in Superconductivity XV. Part II.

[34] R McDermott, M G Vavilov, B L T Plourde, F K Wilhelm, P J Liebermann, O A Mukhanov, and T A Ohki. 2018. Quantum–classical interface based on single flux quantum digital logic. *Quantum Science and Technology* 3, 2 (jan 2018), 024004. https://doi.org/10.1088/2058-9565/aaa3a0

[35] George H. Mealy. 1955. A Method for Synthesizing Sequential Circuits. *The Bell System Technical Journal* 34, 5 (Sept. 1955), 1045–1079. https://doi.org/10.1002/j.1538-7305.1955.tb03788.x

[36] O.A. Mukhanov, S.V. Rylov, D.V. Gaidarenko, N.B. Dubash, and V.V. Borzenets. 1997. Josephson output interfaces for RSFQ circuits. *IEEE Transactions on Applied Superconductivity* 7, 2 (1997), 2826–2831. https://doi.org/10.1109/77.621825

[37] Mustafa Munir, Aswin Gopikanna, Arash Fayyazi, Massoud Pedram, and Shahin Nazarian. 2021. QMC: A Formal Model Checking Verification Framework For Superconducting Logic. In *Proceedings of the 2021 on Great Lakes Symposium on VLSI* (Virtual Event, USA) *(GLSVLSI '21)*. Association for Computing Machinery, New York, NY, USA, 259–264. https://doi.org/10.1145/3453688.3461522

[38] Louis C. Müller and Coenrad J. Fourie. 2014. Automated State Machine and Timing Characteristic Extraction for RSFQ Circuits. *IEEE Transactions on Applied Superconductivity* 24, 1 (2014), 3–12. https://doi.org/10.1109/TASC.2013.2284834

[39] L. C. Müller and C. J. Fourie. 2014. Automated State Machine and Timing Characteristic Extraction for RSFQ Circuits. *IEEE Transactions on Applied Superconductivity* 24, 1 (2014), 3–12. https://doi.org/10.1109/TASC.2013.2284834

[40] Laurence W. Nagel. 1975. *SPICE2: A Computer Program to Simulate Semiconductor Circuits.* Ph. D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/1975/9602.html

[41] Travis E. Oliphant. 2007. Python for Scientific Computing. *Computing in Science Engineering* 9, 3 (2007), 10–20. https://doi.org/10.1109/MCSE.2007.58

[42] S. Polonsky, P. Shevchenko, A. Kirichenko, D. Zinoviev, and A. Rylyakov. 1997. PSCAN'96: new software for simulation and optimization of complex RSFQ circuits. *IEEE Transactions on Applied Superconductivity* 7, 2 (1997), 2685–2689. https://doi.org/10.1109/77.621792

[43] S V Polonsky, V K Semenov, and P N Shevchenko. 1991. PSCAN: Personal Superconductor Circuit Analyser. *Superconductor Science and Technology* 4, 11 (Nov. 1991), 667–670. https://doi.org/10.1088/0953-2048/4/11/031

[44] Lieze Schindler. 2021. *The Development and Characterisation of a Parameterised RSFQ Cell Library for Layout Synthesis.* Ph. D. Dissertation. Stellenbosch University.

[45] Klaus Schneider. 2001. *A Verified Hardware Synthesis of Esterel Programs.* Springer US, Boston, MA, 205–214. https://doi.org/10.1007/978-0-387-35409-5_20

[46] Pavel Shevchenko. [n. d.]. PSCAN2. http://pscan2sim.org/. Accessed: 2021-10-22.

[47] Igor I Soloviev, Nikolay V Klenov, Sergey V Bakurskiy, Mikhail Yu Kupriyanov, Alexander L Gudkov, and Anatoli S Sidorenko. 2017. Beyond Moore's technologies: operation principles of a superconductor alternative. *Beilstein journal of nanotechnology* 8, 1 (2017), 2689–2710.

[48] I Synopsis. 2009. HSPICE: The gold standard for circuit simulation.

[49] Ramy N. Tadros, Arash Fayyazi, Massoud Pedram, and Peter A. Beerel. 2020. SystemVerilog Modeling of SFQ and AQFP Circuits. *IEEE Transactions on Applied Superconductivity* 30, 2 (2020), 1–13. https://doi.org/10.1109/TASC.2019.2957196

[50] R. N. Tadros, A. Fayyazi, M. Pedram, and P. A. Beerel. 2020. SystemVerilog Modeling of SFQ and AQFP Circuits. *IEEE Transactions on Applied Superconductivity* 30, 2 (2020), 1–13. https://doi.org/10.1109/TASC.2019.2957196

[51] Georgios Tzimpragos, Advait Madhavan, Dilip Vasudevan, Dmitri Strukov, and Timothy Sherwood. 2019. Boosted Race Trees for Low Energy Classification. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 215–228. https://doi.org/10.1145/3297858.3304036

[52] Georgios Tzimpragos, Dilip Vasudevan, Nestan Tsiskaridze, George Michelogiannakis, Advait Madhavan, Jennifer Volk, John Shalf, and Timothy Sherwood. 2020. A Computational Temporal Logic for Superconducting Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 435–448. https://doi.org/10.1145/3373376.3378517

[53] Georgios Tzimpragos, Jennifer Volk, Dilip Vasudevan, Nestan Tsiskaridze, George Michelogiannakis, Advait Madhavan, John Shalf, and Timothy Sherwood. 2021. Temporal Computing With Superconductors. *IEEE Micro* 41, 3 (2021), 71–79. https://doi.org/10.1109/MM.2021.3066377

[54] Georgios Tzimpragos, Jennifer Volk, Alex Wynn, James E. Smith, and Timothy Sherwood. 2021. Superconducting Computing with Alternating Logic Elements. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 651–664. https://doi.org/10.1109/ISCA52012.2021.00057

[55] 2020. IEEE Standard for Universal Verification Methodology Language Reference Manual. *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)* (2020), 1–458. https://doi.org/10.1109/IEEESTD.2020.9195920

[56] 2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), 1–590. https://doi.org/10.1109/IEEESTD.2006.99495

[57] S. R. Whiteley. 1991. Josephson junctions in SPICE3. *IEEE Transactions on Magnetics* 27, 2 (1991), 2902–2905. https://doi.org/10.1109/20.133816

[58] Inc. Whiteley Research. [n. d.]. WRspice. http://wrcad.com. Accessed: 2021-10-22.

[59] A. D. Wong, K. Su, H. Sun, A. Fayyazi, M. Pedram, and S. Nazarian. 2019. VeriSFQ: A Semi-formal Verification Framework and Benchmark for Single Flux Quantum Technology. In *20th International Symposium on Quality Electronic Design (ISQED)*. IEEE, Santa Clara, CA, USA, 224–230. https://doi.org/10.1109/ISQED.2019.8697701

[60] Q. Xu, C. L. Ayala, N. Takeuchi, Y. Yamanashi, and N. Yoshikawa. 2016. HDL-Based Modeling Approach for Digital Simulation of Adiabatic Quantum Flux Parametron Logic. *IEEE Transactions on Applied Superconductivity* 26, 8 (2016), 1–5. https://doi.org/10.1109/TASC.2016.2615123