PyLSE: A Pulse-Transfer Level Language for Superconductor Electronics

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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ACM Reference Format:

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 (Python Language for Superconductor Electronics). 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 of our PyLSE language. This FSM abstraction allows the description of
CMOS: Steady voltage levels encode information; thus, wires are considered stateful and the gates stateless.

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.

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 $t_1$ or setup time $t_2$ are erroneous. Assuming these timing violations do not occur, a pulse is produced some propagation delay $t_3$ 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 ($\text{CLK}^\text{hole}/\{Q_{\text{prop}}\}/*_{\text{setup}}$).

$^2$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 $A$ Triggers the transition from the source to dest state. This transition has priority $N$ over other simultaneously-triggered transitions originating from source and takes $t_{\text{tran}}$ 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 $t_{\text{fire}}$ time units later. Finally, according to the Past Constraints, if it’s been less than $t_{\text{dist}}$ since the last time an input $B$ was received during a previous transition, it is an error. $A^N_{\text{tran}}$ is shorthand for $A^N_{\text{tran}}/\emptyset/\emptyset$.

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 $t_{\text{hold}}$ time units. Moreover, to model the hold time constraint of Figure 3, we set $t_{\text{tran}} := t_{\text{hold}}$ and therefore consider the arrival of any other input pulse during this transitional 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 $t_{\text{hold}}$ and using the past constraints, we can model the $t_{\text{setup}}$ time. Firing delay directly models the propagation delay $t_{\text{prop}}$. 
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 \( \{ q \} \) indicates that an output pulse will be emitted during this state transition. The time that it will take for this pulse to appear is \( t_{\text{prop}} \) time units. Therefore, we can use the edge’s firing delay to model the cell’s propagation delay; e.g., by setting \( t_{\text{fire}} = t_{\text{prop}} \).

**Constraints on Past:** The **Past Constraints** portion is used to model the **setup time** constraint; e.g., by setting \( t_{\text{dist}} := t_{\text{setup}} \). In the provided example, any input pulse (indicated by the *) that appears within \( t_{\text{setup}} \) 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 \( M = \langle Q, q_{\text{init}}, \Sigma, \Lambda, \delta, \mu, \theta \rangle \), where

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

We write \( M.\Sigma \) to extract \( \Sigma \), and likewise \( M.\Lambda \) for \( \Lambda \).

The first three domains — \( Q, \Sigma, \) and \( \Lambda \) — are similar to a typical Mealy machine definition. The transition function \( \delta \) 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, \( \mu \), maps tuples of states and inputs to sets of tuples consisting of output symbols and the time they take to appear (i.e. a firing delay). The past constraints function \( \theta \) maps the current state and input to an 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 \( \kappa \in K = Q \times \mathbb{R} \times (\Sigma \to \mathbb{R}) \), parameterized over a current state \( q \in Q \), a real-valued time \( t_{\text{done}} \), and a mapping \( \Theta : \Sigma \to \mathbb{R} \) that associates each input with the last time it was seen. This is written as \( k(q,t_{\text{done}},\Theta) \), with the \( t_{\text{done}} \) being used to represent the end of the unstable period during which time the machine is transitioning. The initial configuration is \( k^M_{\text{init}} = k(q_{\text{init}},0,\sigma \mapsto \infty|\sigma \in M.\Sigma) \).

**Transition Relation.** Given the current configuration \( k(q_{\text{curr}},t_{\text{curr}},\Theta) \), the Transition Relation is interpreted as follows. If the machine receives an input \( \sigma \) at time \( t_{\text{curr}} \) and it has been long enough to have finished entering state \( q_{\text{curr}} \) (i.e. \( t_{\text{curr}} \geq t_{\text{done}} \)), it proceeds to a new configuration \( k(q_{\text{next}},t_{\text{done}}^\sigma,\Theta) \). It does so by remembering (1) the next state \( q_{\text{next}} \), (2) the time at which the new transition should be completed \( t_{\text{done}}^\sigma = t_{\text{curr}} + t_{\text{arr}} \), and (3) the time it saw the current input, via \( \Theta' = \Theta[\sigma \mapsto t_{\text{arr}}] \) (see NORMAL-\( \kappa \)). Otherwise, if it is not yet ready to receive inputs because \( t_{\text{curr}} < t_{\text{done}}^\sigma \) (see ERROR-\( \kappa \) TRAN) or because any input \( \sigma' \) was received less than \( \theta(q,\sigma') + t_{\text{dist}} \) ago (see ERROR-\( \kappa \) CONS), it proceeds to the special \( q_{\text{err}} \) state. \( q_{\text{err}} \) is the target state of any transition whose timing conditions can’t be satisfied.

**Dispatch and Trace Relation.** The Dispatch Relation enables the machine to continue processing inputs. It works by retrieving the highest priority transition that leaves \( q_{\text{curr}} \) for all the inputs \( \sigma \) in the set of simultaneous inputs \( \sigma \) arriving at \( t_{\text{curr}} \). 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 **network**, is a tuple \( C = \langle \bar{M}, \bar{w}, \Sigma, \Lambda \rangle \) composed of a set of PyLSE Machines \( \bar{M} \) (accessed as \( C.\text{machines} \)), a set of connective wires \( \bar{w} \) (accessed as \( C.\text{wires} \)), and a set of circuit inputs \( C.\Sigma \) and outputs \( C.\Lambda \). A wire is a tuple \( w = \langle a, \beta \rangle \) such that \( a \in M'.\Lambda \cup C.\Sigma \) and \( \beta \in M''.\Sigma \cup C.\Lambda \) for some \( M', M'' \in \bar{M} \).

**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 \( k^C_{\text{init}} \) composed of (1) all individual PyLSE Machine initial configurations \( \kappa \) and (2) a list of input pulses \( ps \) tagged with the wires where they are headed, i.e. \( k^C_{\text{init}} = \langle \kappa, ps \rangle \), where \( \kappa = \{ k^M_{\text{init}}|M \in C.\text{machines} \} \) and \( ps = \{ (\sigma', t_{\text{arr}})|\langle \sigma', t_{\text{arr}} \rangle \in ts \land (\sigma', \sigma'' \in C.\text{wires} \} \). The network proceeds until there is no more work to do, such that \( k^C_{\text{init}} + ps \rightarrow_{\text{net}} k^C + ps' \). In other words, all pending pulses in \( ps \) are directed toward the circuit output.
We use the above PyLSE Machine formalism to develop a UCSBarchlab/PyLSE 
3 Transitional nant logic scheme for SCE designs, the ability to easily de-
the barrier of entry for new users and gain the productivity 
practical embedded DSL that eases the description and anal-
ous pending pulses on the heap 
\( p_s \)
Nondeterminism occurs when there are multiple simultane-
\( p_s \)
amapping where 
\( \pi \)
and 
\( \sigma \)
are heaps of pulses, we use 
\( x \)
\( \delta(\sigma_{curr}, \sigma') \)
\( \Theta(\sigma \mapsto \tau) \)
produces an updated mapping where \( \sigma \) now maps to \( \tau \). We use \( S[y/x] \) to denote \( y \) replacing \( x \) in \( S \). The helper function \( \text{getSimPulses} \) extracts the pulse heap \( p_s \) 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 \( p_s \) going to different PyLSE Machines; the helper function \( \text{getSimPulses} \) 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. 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
\( pt \in \text{Port} \)  \( st \in \text{State} \)  \( \sigma \in \text{Store} \)  \( e \in \text{Exp} \)  \( n \in \mathbb{Z} \)  \( \tau \in \text{Time} \)
\( p \in \text{Program} \)  \( \text{ins} \)  \( pt^+ \)  \( \text{outs} \)  \( pt^+ \)  \( \text{cells} \)  \( cell^+ \)  \( \text{conns} \)  \( con^+ \)
\( cell \in \text{Cell} \)  \( \text{Cell} \Rightarrow pm \mid h \)
\( pm \in \text{PyLSEMachine} \)  \( \text{states} \)  \( st^+ \)  \( \text{start} \)  \( st \)
\( \text{ins} \)  \( pt^+ \)  \( \text{outs} \)  \( pt^+ \)  \( \text{edges} \)  \( ed^+ \)
\( h \in \text{Hole} \)  \( \text{ins} \)  \( pt^+ \)  \( \text{outs} \)  \( pt^+ \)  \( \text{func} \)  \( \lambda pt^+ \sigma \tau.e \)
\( ed \in \text{Edge} \)  \( \text{priority} \)  \( n \in \text{src} \)  \( st \)  \( \text{dest} \)  \( st \)  \( \text{trigger} \)  \( pt \)
\( \text{transitime} \)  \( \tau \)  \( \text{firing} \)  \( \mu \)  \( \text{constraints} \)  \( \text{\theta} \)
\( con \in \text{Connection} \)  \( \Rightarrow m.pt \leftarrow m.pt \)
\( m \in \text{Entity} \)  \( \Rightarrow \text{cell} \mid p \)
\( \mu, \theta \in \text{TimingMap} \)  \( \Rightarrow [pt \mapsto \tau]^+ \)

**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. \( \pi_i (\ldots, x_i, \ldots) = x_i \) is standard tuple projection. \( \Theta(\sigma \mapsto \tau) \) produces an updated mapping where \( \sigma \) now maps to \( \tau \). We use \( S[y/x] \) to denote \( y \) replacing \( x \) in \( S \). The helper function \( \text{getSimPulses} \) extracts the pulse heap \( p_s \) 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.

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

\( \text{\theta} \text{\_TimingMap} \Rightarrow [pt \mapsto \tau]^+ \)
At this level, pure Python code is wrapped in a specialized 
cate via pulses with the rest of the system. The 
allowing non-transition-based abstract "holes" to communi-
Functional 
interface (by implementing a 
the need to describe every single block via interacting tran-
ins = [ 
{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. Transi-
tions 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 values 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 prototyp-
ing and exploration of more complicated designs without 
the need to describe every single block via interacting tran-
sition 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.

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

Figure 9. An example Functional (“hole”) element model-
ing a memory with 16 addresses, each storing 2 bits.

Figure 10. Graphical results of simulating the memory 
Functional class.
whose output is delayed via a Josephson transmission line. This encapsulation enables basic cells to resemble Python objects and return one or more output wire objects as result. Functions introduced temporal conventions \[ 52\]. Calling the function \( \text{min}\_\text{max}(a, b) \): def \( \text{min}\_\text{max}(a, b) \):

\[
\begin{align*}
\text{a} & = \text{c}\_\text{inv}(a) \\
\text{b} & = \text{c}\_\text{inv}(b) \\
\text{low} & = \text{c}\_\text{inv}(a, b) \\
\text{high} & = \text{c}\_\text{inv}(a, b) \\
\text{return low, high}
\end{align*}
\]

This function, \( \text{memory()} \), takes in twelve boolean-valued arguments and a thirteenth argument, \( \text{time} \), which is implicitly passed as the last argument to all functional elements. The read and write addresses, \( r\_\text{addr} \) and \( w\_\text{addr} \), are split into four 1-bit inputs. A pair of nonlocal variables \( \text{raddr} \) and \( \text{waddr} \) 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 \( \text{raddr} \) and \( \text{waddr} \) are reset, ready for the next period. The arguments are internally connected to \( \text{PyLSE Wires} \) 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 \( \text{memory()} \), 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 \footnote{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.} following recently introduced temporal conventions \[ 52\]. Calling the function \( \text{min}\_\text{max}(a, b) \) causes its constituent cells and connective wires to be instantiated via the calls to the encapsulating functions \( \text{s}, \text{c}, \text{c}\_\text{inv}, \) and \( \text{jtl} \). These functions take in wire objects and return one or more output wire objects as result. This encapsulation enables basic cells to resemble Python 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 \texttt{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...
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 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 propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed into the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get push...
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 $c_h$, $c_p$, and $c_\text{clk}$ are compared against the past constraint value, $\tau_{\text{setup}}$, in the first edge’s guard. The TA goes to an error state if these constraints are violated and otherwise transitions to $q_b$. Figure 14b is the result of this first conversion.

To detect inputs while in the transitional period, Figure 14c inserts three additional states – $\text{err}_a$, $\text{err}_b$, and $\text{err}_\text{clk}$ – 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 $\tau_{\text{prop}}$ 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.
Claim 3. PyLSE can be used in conjunction with a state-of-the-art model checker to formally verify properties of basic \( \text{SCE cells and larger designs.} \)

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 \( \text{SCE cells using pulse-based signaling and systems built out of such cells.} \)

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 \( \text{SCE cell as a distribution.} \)

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

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

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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Schematic (Cadence)</th>
<th>PyLSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines</td>
<td>Time (s)</td>
<td>Size</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>C</td>
<td>81</td>
<td>2.840</td>
</tr>
<tr>
<td>InvC</td>
<td>87</td>
<td>2.987</td>
</tr>
<tr>
<td>Min-Max Pair</td>
<td>140</td>
<td>4.608</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td>250</td>
<td>52.565</td>
</tr>
</tbody>
</table>

Figure 15. A eight-input bitonic sorter, composed of twenty-four comparators (see Figure 11a). It takes eight individual input wires \((i_0\text{ through }i_7)\), which are produced in arrival time order, after some network delay, on \(o_0\text{ through }o_7\).
network, i.e. $22 \times 6 = 132$ ps. The PyLSE design, with a total delay of $25 \times 6 = 150$, 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 $16.6 \times$ smaller than their circuit schematic counterparts and take an average $9879 \times$ 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:

```python
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:

```python
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 $O_0$, followed by $O_1$, etc. until the last output $O_N$ for some power-of-two $N$:

```python
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[i] <= y[i] 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\})) &&
(firingauto4.fta_end imply (\{global == 890\} || \{global == 3290\}))) &&
(firingauto5.fta_end imply (\{global == 890\} || \{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, errTAs and errTAs, 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>PyLSE</th>
<th>UPPAAL</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td>Cells</td>
<td>States</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>InvC</td>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>M</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JTL</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>And</td>
<td>11</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Or</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Nand</td>
<td>12</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Nor</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Xor</td>
<td>1</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>Xnor</td>
<td>12</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Inv</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>DRO</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>DRO SR</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>DRO C</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2x2 Join</td>
<td>20</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Min-Max</td>
<td>5</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>Race Tree</td>
<td>16</td>
<td>18</td>
<td>32</td>
</tr>
<tr>
<td>Adder (Sync)</td>
<td>13</td>
<td>19</td>
<td>33</td>
</tr>
<tr>
<td>Adder (xSFQ)</td>
<td>31</td>
<td>83</td>
<td>121</td>
</tr>
<tr>
<td>Bitonic Sort 4</td>
<td>6</td>
<td>30</td>
<td>54</td>
</tr>
<tr>
<td>Bitonic Sort 8</td>
<td>24</td>
<td>120</td>
<td>216</td>
</tr>
</tbody>
</table>

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

```plaintext
A[] not c0.C_r_err_a_1 || c0.C_r_err_a_11 || c0.C_r_err_a_16 || ...
```

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.

**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


