Citrus: Algebraic Reasoning about Superconductor Electronics

HARLAN KRINGEN, University of California at Santa Barbara, USA
TIMOTHY SHERWOOD, University of California at Santa Barbara, USA
BEN HARDEKOPF, University of California at Santa Barbara, USA

We present Citrus, an embedded DSL in the dependently-typed language Agda that formalizes high-level abstractions for specifying and reasoning about superconducting electronics (SCE) circuits. We build on the existing PyLSE language, a Python DSL for writing SCE programs that provides facilities for simulating designs and for verification via model-checking (by compiling its basic structures into Timed Automata). Citrus expands on the verification capabilities of PyLSE by defining equivalence over SCE gates, as well as corresponding equational reasoning lemmas, and providing a toolbox of functional combinators for designing and analyzing larger circuits. The formalization enables a large increase in expressivity, specifically in the form of *algebraic* reasoning about SCE designs. We evaluate Citrus using two sets of case studies. In the first, we establish several equational laws which are often used by SCE designers but have yet to be formally proven. In the second, we prove a verification task which the PyLSE language was unable to prove due to the state space explosion inherent in its model checking approach. In this way, Citrus provides a simple functional programming language, equivalent to PyLSE, with increased verification capabilities.

CCS Concepts: • **Theory of computation** → **Timed and hybrid models**; **Interactive computation**; **Logic and verification**; • **Hardware** → **Emerging tools and methodologies**; **Emerging languages and compilers**; *Emerging architectures*; *Emerging simulation.*

Additional Key Words and Phrases: functional reative programming, coinductive type theory, superconductor electronics

ACM Reference Format:
Harlan Kringen, Timothy Sherwood, and Ben Hardekopf. 2026. Citrus: Algebraic Reasoning about Superconductor Electronics. *Proc. ACM Program. Lang.* 10, ICFP, Article 292 (August 2026), 26 pages. https://doi.org/10.1145/3828690

1 Introduction

Superconducting electronics (SCE) are an emerging technology that promises ultra power-efficient circuits. However, SCE operate on asynchronous, pulse-based logic rather than the traditional synchronous, level-based Boolean logic of CMOS circuits that are the most common implementation of modern commercial electronics. Information in a SCE circuit is transmitted via near-instantaneous pulses, with the relative timing of pulses on different wires conveying semantic meaning. SCE as a technology is under active development and there are no established formalisms for describing or reasoning about SCE circuits. Most commonly, SCE circuits are described via low-level physical circuit diagrams in a tool such as SPICE [Delport et al. 2019]. Recent work on the PyLSE language [Christensen et al. 2022] proposed a high-level Hardware Description Language (HDL) abstraction for describing SCE circuits, based on a timed version of Mealy Machines [Rutten 2019]

Authors' Contact Information: Harlan Kringen, University of California at Santa Barbara, Santa Barbara, USA, kringen@ucsb.edu; Timothy Sherwood, University of California at Santa Barbara, Santa Barbara, USA, sherwood@cs.ucsb.edu; Ben Hardekopf, University of California at Santa Barbara, Santa Barbara, USA, ben.hardekopf@gmail.com.

ACM 2475-1421/2026/8-ART292
https://doi.org/10.1145/3828690

called *PyLSE Machines*.[1] That work gave PyLSE Machines an operational semantics as a state transition system and a denotational semantics via translation to Timed Automata [Alur and Dill 1994]. These semantics allow PyLSE Machine descriptions of SCE circuits to be directly simulated and, by translation to Timed Automata, to be model checked using existing tools such as UPPAAL [Bengtsson and Yi 2004].

However, while PyLSE is a useful tool for describing and simulating SCE circuits, it does not allow for *reasoning* about such circuits at the PyLSE Machine level of abstraction. For example, we cannot reason about the equivalence of two PyLSE Machines, or whether a particular transformation on PyLSE Machines (e.g., to optimize an SCE circuit) is guaranteed to preserve equivalence. **The main contribution of this paper is to provide a framework for algebraically reasoning about PyLSE Machines.** Our framework is inspired by the well-known concept of *functional arrows* [Hughes 2005], which have been applied in fields as diverse as graphics [Elliott and Hudak 1997a] and robotics [Hudak et al. 2003]; we show that they can also be applied to the domain of SCE to create a rich, expressive language for algebraically reasoning about PyLSE Machines. We formalize our framework in a tool called Citrus, implemented using the dependently-typed language Agda [Bove et al. 2009], and demonstrate its usefulness via a number of case studies to prove SCE equivalencies that PyLSE cannot establish (and, in fact, that have never been proven before).

The key insights behind Citrus are that (1) Mealy Machines, as transducers, describe relations between sets; (2) we can describe these relations independently from Mealy Machines in a well-typed way inspired by *arrows* [Hughes 2005]; and (3) we can specifically use a structure akin to *looped arrows* [Liu et al. 2009] to incorporate and track the necessary state and timing information required by PyLSE Machines (which augment Mealy Machines with timing information). These time-tracking looped arrows are regular arrows augmented with an additional abstraction for tracking timing that is necessary for specifying the timing-sensitive nature of SCE. We call the Citrus data structure describing a particular combination of timing-sensitive arrows a `PulseGate`. In Citrus an SCE circuit, using the PyLSE Machine abstraction, is described in a functional, well-typed way as a composition of `PulseGates` allowing it to respect the inherently timed and stateful nature of SCE gates.

Given two Citrus programs we can equationally reason about their equivalence according to multiple types of bisimulation. We concentrate in this paper specifically on adapting *time-abstract weak bisimulation* [Aceto et al. 2007]. A weak bisimulation means that the two programs are allowed to take differing numbers of steps; time-abstract similarly means that the two programs are allowed to take differing amounts of time (i.e., the circuit time delays can be different). We argue that this notion of equivalence is well-suited for reasoning about SCE circuits. Intuitively, since we want to reason about circuit equivalence to verify optimizations (which inherently change circuit timing) a strong notion of timing bisimulation that doesn't allow for different timing characteristics doesn't make sense. Time-abstract bisimulation is suitable because SCE circuits are designed so that a specific sequence of inputs will cycle the circuit through a set of states and then back to the original state. In this case, "state" doesn't refer to a specific node in an automaton, but simply whatever data is being kept track of in a feedback loop. An SCE circuit begins in some initial configuration in which it is ready to receive inputs. It then receives pulses, after which it may be unable to receive new inputs, for instance if it is delaying or waiting for more inputs. Upon satisfying its remaining conditions, it then returns to its original configuration. Two time-abstract weakly bisimilar Citrus programs are guaranteed to have the same behavior on the same input sequences, which is enough to establish a useful notion of functional SCE circuit equivalence. We

[1] Pronounced "pulse machine".

adapt this notion of equivalence to our `PulseGates` and their unique notion of cycling. These proofs are all mechanized via Agda's dependent type system.

Citrus takes advantage of two formalisms: coinduction, in the form of a coinductive types system (in Agda), and Arrows, from Functional Reactive Programming. Citrus itself is not an implementation of or extension to coinductive type theory. We claim that arrows and coinduction are aptly suited to compositionally, and more generally, model SCE. Citrus is the implementation that allows arrows to describe SCE and moreover for coinduction to describe the correctness properties, such as circuit equivalence, which are features specifically lacking or impossible in previous approaches.

The contributions of this paper are:

- (Section 3) We describe Citrus, a dependently-typed framework embedded in Agda for describing SCE circuits as PyLSE Machines that allow for algebraic reasoning.
- (Section 4) We explain how to reason equationally about the equivalence of Citrus programs.
- (Section 5) We demonstrate the utility of Citrus on a number of case studies, divided into two groups: (1) a set of real-world circuit optimizations used by SCE designers that are believed, but not previously proven, to be semantics-preserving; and (2) a non-trivial SCE circuit taken from the PyLSE paper [Christensen et al. 2022], showing that Citrus can describe the same SCE circuits as PyLSE and allows for reasoning about them in a way that PyLSE does not.

We provide necessary background information in Section 2. We describe related work in Section 6 and conclude in Section 7.

2 Background

In this section we provide background on SCE (Section 2.1), PyLSE (Section 2.2), functional arrows (Section 2.3), and Agda (Section 2.4).

2.1 Superconducting Electronics (SCE)

At the hardware level, superconducting elements can be characterized according to three main features (1) zero resistance in static circuits at superconducting temperatures, (2) the manipulation of the Josephson effect as the main switching element and (3) the propagation of signals (single flux quanta) at picosecond durations and millivolt amplitudes [Tzimpragos et al. 2021]. At the software level we can understand this as a unique information encoding, with features contrary to traditional semiconductor electronics (CMOS). In CMOS, the information encoding is based on static voltage levels which may be mapped onto true/false values in the manner of Boolean logic. SCE encodes information according to instantaneous pulses with almost no voltage draw. Theoretically, this would allow for extremely fast circuits with very low power consumption [Holmes et al. 2015]. More broadly put, at the software level, SCE is interesting because it is a technology that can address post-Moore scaling issues [Soloviev et al. 2017]. In this way, SCE is interesting on two accounts, for its unique information encoding and for the possibility of creating highly performant electronics.

Although our present work abstracts much of the hardware complexity of superconducting electronics, we provide here a brief explanation of the basic principles of SCE. In SCE, a "flux quantum" carries the unit of information. In practice, these are *pulses* which occur at very low temperatures [Hayakawa et al. 2004]. Pulses last on the order of one picosecond, allowing for subterahertz frequencies. This technology provides for high-throughput circuits with power consumption three orders of magnitude lower than conventional semiconductor-based circuitry (CMOS). SCE gates are conceptually similar to CMOS gates in some ways, but differ in others. Gates are functional

objects which are connected by wires on which pulses arrive and depart. The main difference SCE gates have from CMOS gates is that, while the pulses themselves are instantaneous and transient, the gates are stateful. That is, the gates respond to pulses by internally changing their state. SCE gates can then keep track of when pulses arrived and reset to an initial configuration, e.g., after some time has passed or when another input has arrived.

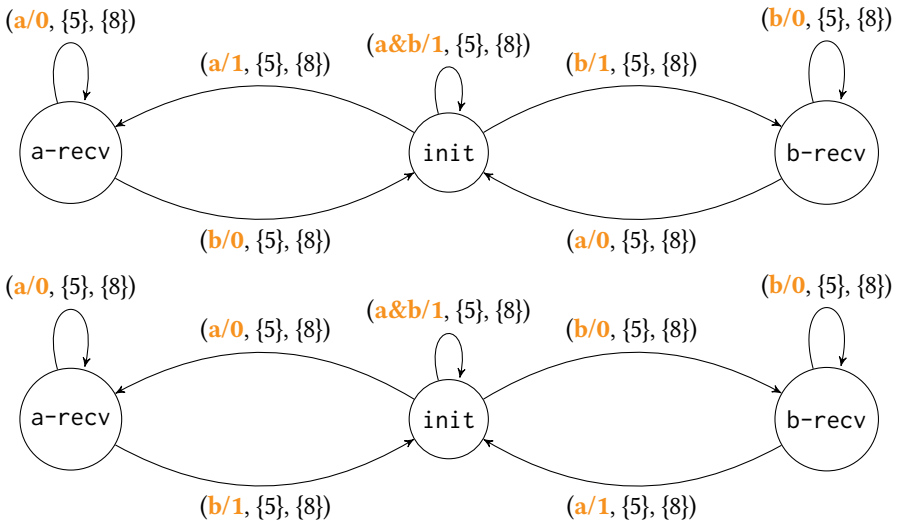

Fig. 1. First Arrival (FA) and Last Arrival (LA) SCE gates as timed Mealy Machines, with FA on the top and LA on the bottom. The syntax "(a & b)/1" covers the case that the "a" and "b" pulses arrive simultaneously. The I/O information is colored in orange while the timing information, colored black, is given as a tuple next to it. The tuple includes the transition delay and the firing delay. Notationally, the tuple "(a/1, {5}, {8})" indicates an "a" pulse arrives, triggers a delay of 5 picoseconds as the gate transitions, and then triggers a pulse as an output which results in a 8 picosecond firing delay. An "a/0" means that no pulse is triggered.

Figure 1 describes two SCE gates, First Arrival (analogous to a conventional OR gate) and Last Arrival (analogous to a conventional AND gate) using Mealy Machines (i.e., a finite state transducer) augmented with timing information. An FA gate pulses when it receives the first of two possible signals and resets once it has received the second; an LA gate pulses when it receives the second of two possible signals and immediately resets.

A Mealy Machine can be represented as a function of type $(Input \times State \rightarrow Output \times State)$. The notation $x/1$ on an edge means the gate receives a pulse on the x wire (or, in the theoretical model, reads an x symbol from the input) and then sends a pulse on the output wire (writes a 1 symbol to the output); the notation $x/0$ means the gate reads a pulse on the x wire and *doesn't* send a pulse on the output wire (indicated in the theoretical model by writing a 0 symbol to the output).

The First Arrival (FA) gate begins at the *Init* state and waits to receive either an a or b pulse. If it receives an a pulse it fires, that is, sends the output pulse 1. After this event it waits for the opposite pulse, in this case the b pulse, in the $a - recv$ state. While waiting, if it receives another a pulse it outputs 0, indicating that it does *not* output a pulse, and stays in the $a - recv$ state. If it sees a b pulse while in the $a - recv$ state it outputs a 0 (no pulse) and returns to *Init*, i.e., it resets to its beginning configuration. The case is symmetric for beginning with a b pulse. Note again that pulses are associated with wires, that is, an a pulse means a pulse received on the a wire and a b

pulse means a pulse received on the b wire; pulses themselves are indistinguishable beyond what wire they are on and when they arrived.

The Last Arrival (LA) gate is largely dual. After receiving an initial pulse, it does not output a pulse and instead begins waiting for the opposite pulse. Only after receiving *both* pulses does it fire and return to the *Init* state. If we consider only the orange I/O information in Figure 1, we get the simple, untimed description of these gates. The additional tuples of numbers labeling the transitions indicate some of the timing information that is associated with all SCE gates. Upon receiving a pulse, for instance, the FA gate spends 50 picoseconds before updating its internal state, and there is a delay of 120 picoseconds before the output pulse appears at the receiving gate downstream.

We will focus on two aspects of timing information common to all SCE gates, namely, transition delay and firing delay. Transition delay refers to the amount of time a gate spends before it reconfigures its internal state. For instance, a pulse may arrive at a gate, which then propagates the pulse, but then spends 50 picoseconds transitioning before returning to its original configuration. A pulse arriving during this transition delay would be ignored by the gate and possibly indicates an error in the design. If a gate propagates, or "fires", a pulse, there is also a certain amount of time spent before the pulse appears to whatever downstream gate receives it, called the firing delay.

2.2 PyLSE

The PyLSE language defines two main representations for dealing with SCE: The first is the PyLSE Machine which is an extension of the basic Mealy Machine to include the variety of timing information needed to describe hardware superconductors; the second is the timed automaton derived from a given PyLSE Machine. The first representation is useful for rapid prototyping and simulation while the second representation is geared to model checking. We reproduce an example of the Last Arrival circuit in PyLSE in Listing 1. The states of a PyLSE Machine are abstract states, while the majority of the information resides on the edges. The information on each edge is a 3-tuple consisting of "Trigger" information, "Firing Outputs" and "Past Constraints". Triggers and Outputs have names and durations (transition times and delays, respectively). Past Constraints define minimum times required until a new pulse may be ingressed. Designing larger circuits in the PyLSE language is done procedurally by assigning outputs of circuit elements to temporary variables which are then used as inputs to other circuit elements.

2.3 Arrows and Functional Reactive Programming

The automata-theoretic presentation of SCE is useful for description and simulation of SCE circuits, but not for reasoning about those circuits. An alternative presentation is to find a structure that captures the same information but is more functional and compositional. To this end, we adapt the arrow type [Hughes 2000] prominently used in functional reactive programming (FRP). Arrows may be thought of as generalized functions, that is, they are types which record the domain and codomain at the type level.

Functions naturally constitute an instance of arrows, but so do other structures such as stream transformers, Moore Machines, Mealy Machines, and many others [Paterson 2003]. This unified type signature allows us to define combinators, or higher order functions, which describe various kinds of compositions including sequential and parallel composition, branching choice, and even value-level recursion. In this way, arrows may be thought of as a language for dataflow graphs. These features can be bundled together into a common typeclass such as one might find in Haskell. However, arrows must obey certain laws for any types they are applied to, such as preserving identities, associativity, and others; these laws allow us to reason about arrow compositions algebraically.

The Arrow typeclass requires a few laws to hold on the given type and combinators. This is entirely analogous to the Monad typeclass which requires verifying, e.g. that the $returnx = f = fx$.

```
class LA(SFQ):
    ''' Last Arrival Gate
    '''
    name = 'LA'
    inputs = ['a', 'b']
    outputs = ['q']
    firing_delay = 8.0
    transitions = [
       {'id': '0', 'source': 'idle', 'trigger': ['a', 'b'], 'dest': 'idle',
         'priority': 0},
       {'id': '1', 'source': 'idle', 'trigger': 'a', 'dest': 'a_arrived',
         'priority': 0}
       {'id': '2', 'source': 'idle', 'trigger': 'b', 'dest': 'b_arrived',
         'priority': 0},
        {'id': '3', 'source': 'a_arrived', 'trigger': 'b', 'dest': 'idle',
         'firing': 'q', 'transition_time': firing_delay},
        {'id': '4', 'source': 'a_arrived', 'trigger': 'a', 'dest': 'a_arrived'},
        {'id': '5', 'source': 'b_arrived', 'trigger': 'a', 'dest': 'idle',
         'firing': 'q', 'transition_time': firing_delay},
        {'id': '6', 'source': 'b_arrived', 'trigger': 'b', 'dest': 'b_arrived'},
    ]
    jjs = 5
```

Listing 1. The Last Arrival gate as described in PyLSE.

The basic Arrow (and Monad) laws can be found in Hughes' original treatment [Hughes 2000]. For example, composition must be associative $f >>> (g >>> h) = (f >>> g) >>> h$. We also need rules connecting the various combinators. To this end, *arr* must respect composition as $arr(f >>> g) = arrf >>> arrg$. As the basic Arrow class can be extended to include more composition operators, more laws must be added to guarantee the expected computational interpretation. The main extensions for Arrows, including choice combinators and looping, along with their required laws, are given in [Paterson 2003]. The Arrow formalism is less restrictive than the Monad, intuitively focusing more on composition than effects. Arrows have found a wide degree of popularity in the field of Functional Reactive Programming (FRP). As Hudak notes in [Wan and Hudak 2000], FRP is a framework for programming systems in a declarative way. The systems tend to have in common features such as continuously responding to inputs, being sensitive to timing or clocks, and composing components in complex configurations. FRP has been used to describe animation programs [Elliott and Hudak 1997b], robots [Hudak et al. 2003], and music production [Bärenz 2020]. Typically, FRP is used in modeling systems which can be described structurally, and the computational side-effects are relatively minimal.

We will draw inspiration from the Arrow typeclass which consists of four combinators, which are shown graphically in Figure 2 [Paterson 2001]. The lifting combinator *arr f* takes a plain function and lifts it into the Arrow type. The sequence combinator >>> passes the output of one arrow to the next. The *first* combinator takes two inputs but runs the arrow only on the first input. From these components we can build a parallel combinator, ***, which runs one arrow on the first input and one on the second input, and many others. The Arrow type essentially describes stateless, feed-forward dataflow graphs.

However, to use arrows to describe functions that correspond to Mealy Machines, i.e., that have signatures such as $(Input \times State \rightarrow Output \times State)$ which persist and repeatedly respond to inputs, the basic arrow type isn't quite enough. Instead we must take a fixpoint of the type to represent all functions defined with this signature. We use *State* as a recursion variable and rewrite the signature

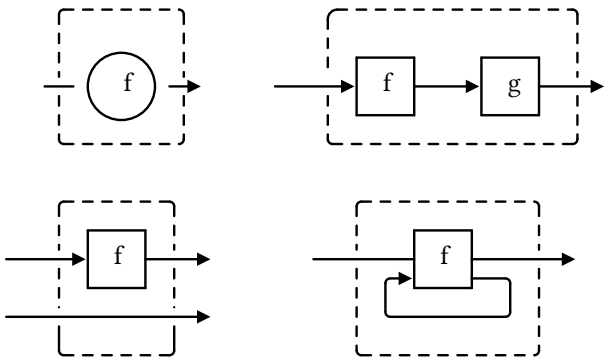

Fig. 2. The Arrow typeclass contains several combinators which allow for the construction of complex "dataflow graph"-like terms. We implement these combinators in Citrus. The first simply lifts a function f into an arrow. The second is the sequence combinator >>>, the third, *first*, runs an arrow only on the first input, forwarding the second, while the fourth, *loop*, is the loop arrow that feeds back information in addition to outputting.

as $F(State) = Input \longrightarrow (Output \times F(State))$. This signature denotes a function that, upon receiving an input, responds with an output and additionally a copy of itself stepped forward "in time". These functions have no termination condition—they keep running forever, and so we use a greatest fixpoint rather than the least fixpoint represented by typical inductive types. Types defined in this way are not *recursive* but instead *corecursive*.

2.4 Agda and Coinduction

Agda is a dependently type programming language [Bove et al. 2009]. Used as a proof assistant (i.e., an interactive theorem prover), it sits in a space similar to Rocq [Paulin-Mohring 2012]—both languages use their dependent type systems as a way to express and prove logical properties. For this work, we concentrate on using dependent types as a way to formalize a model for superconducting circuits. The expressive power of dependent types allows us provide a DSL that can reason about complex forms of equality over circuits. Agda provides convenient facilities for working with coinductive types, as discussed below. Rocq, in contrast, makes coinductive types somewhat more awkward to work with but has greater support for proof automation (e.g., the use of tactics). Either language could be used for Citrus, but we chose to start with Agda because coinductive types are a large part of what we need to reason about SCE. It would be interesting future work to port Citrus to Rocq and compare and contrast the facilities of the two languages.

In Agda, similar to Haskell, the `data` keyword establishes a data type as the least fixpoint of some type-level function. In Haskell the least and greatest fixpoints coincide, and thus `data` declarations suffice for both finite and infinite structures; in contrast for Agda we build coinductive types explicitly using the `record` construct with the keyword `coinductive`. Records may be thought of as generalized products, or tuples, defined by fields. These fields may be used to project out the data contained within the record. The intuition is that, similar to the stream example from above, the defining characteristic of records, or product types, is that we may *destruct* them to observe their values, in contrast to `data` or sum types, which focus on *constructing* terms of the type.

The infrastructure by which Agda's typechecker is able to confirm a coinductive function is productive is based on structures called "sized types" [Abel 2010]. For the programmer, sized types act like annotations to type signatures which help Agda determine *how deeply* a coinductive

function is being observed or deconstructed. It is not necessary for the reader to understand these annotations, but we point them out so as to prevent any confusion in the code instances shown below. There are two main instances of sized types to be aware of. First, we use the ∞ symbol to mark a coinductive type that is "fully defined". That is, when we want to express the equivalence of two circuits, we would like this to hold for circuits defined at all possible depths of observation. Second, we will use $\{i\}$ to denote arbitrary depths of observation and $\{j : \text{Size}< i\}$ to denote arbitrary depths below a given depth.

3 The Citrus Language

Citrus is an embedded language inside Agda, consisting of abstractions and operators aimed at describing SCE circuits as PyLSE Machines. These abstractions and operators are: (1) core data structures for representing timed pulses; (2) the `PulseGate` data structure which represents basic SCE elements; and (3) the composition operators used to compose `PulseGates` into SCE circuits.

3.1 Timed Pulses

To model SCE circuits, we must model infinitesimal pulses occurring at specific timestamps. At a given timestamp we can observe a "pulse", which is either an instantaneous packet of information or its absence. For modeling purposes we add a third possibility, an "error" signal used to model when a pulse arrives at a gate during a time interval in which the gate cannot receive a pulse correctly (and which is propagated as the output of the gate when this event happens). While SCE circuits notionally operate in real-valued time, Citrus uses the natural numbers instead—all time intervals that we deal with are rational, and by choosing the proper units we can describe them using naturals. Pulses in SCE circuits travel on wires, and we abstract these wires as vectors of pulses (with one element per wire). We can now describe our pulse-based I/O representation as shown in Listing 2.

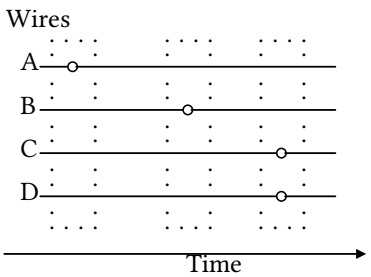

Fig. 3. We treat pulses in Citrus as "vertical" slices of time across a selection of wires. Each slice, indicated by a dotted line, is a vector which is indexed by the number of wires. Here we have three `TimedPulse` 4s over wires A, B, C, D. The open circles indicate pulses.

```
Time : Set
Time = ℕ
data Pulse : Set where
  * : Pulse -- pulse
  ∅ : Pulse -- no pulse
  err : Pulse -- error signal
TimedPulse : (n : ℕ) → Set
TimedPulse n = (Vec Pulse n) × Time
```

Listing 2. The basic definitions for dealing with time in Citrus: time is a Natural number; a `Pulse` is one of of three possible outcomes; a `TimedPulse` is a vector of `Pulses` indexed by the number of wires on which a pulse can arrive.

As the listing shows, a Citrus program will both take and output a vector of pulses at a given timestamp. A `Pulse` itself is one of three possible outcomes: a pulse, nothing, or an error signal. A `TimedPulse` is a vector of `Pulses` indexed by the number of wires on which a pulse can arrive. This "vertical" view of time means that we do not explicitly keep track of a history of pulses. This

representation tracks the operation of SCE more naturally than the alternative because SCE gates themselves keep track of I/O and state information in an asynchronous, non-global manner.

3.2 PyLSE Machines as Coalgebras

The PyLSE Machine is itself based on a type of finite state transducer called a Mealy Machine. Mealy Machines can be thought of as stateful functions, or maps between finite state automata. More formally, Mealy Machines have a straightforward presentation in terms of coalgebras [Rutten 2019]. The theory of coalgebras has been suggested as a natural framework for state-based systems: while terminating functions are usually represented as recursive/inductive objects, state-based, non-terminating systems are represented using corecursion and coinduction instead. The former can be presented as algebras and the latter as coalgebras. We do not require the details of this theory for our purposes and direct the interested reader to [Jacobs 2016]. Fortunately, the theorem prover Agda makes available a wide class of coinductive types using the `record` structure. The basic Mealy Machine coalgebra is interpreted by the coinductive type defined in Listing 3 below.

```
record MealyMachine (I O : Set) : Set where
    coinductive
    field
      resp : I → O
      step : I → MealyMachine I O
```

Listing 3. MealyMachine data structure

The interpretation of Listing 3 as a record is that, given a term of this type, we can *destruct* it using its fields. That is, given a term of type MealyMachine, we can call the `resp` and `step` functions on it. The `resp` function stands for "response" and, given an input of type `I`, will give us an output of type `O`. The `step` function takes an input of type `I` and emits the next step of the object "in time". The keyword `coinductive` tells the type checker that we can do this infinitely. The text can be intuitively restructured as follows: $MealyMachine \to I \to O \times MealyMachine$, i.e., given a MealyMachine and an input, we can generate an output and the next MealyMachine. For Citrus, the inputs I and O should be `TimedPulses`.

The `MealyMachine` type will serve as a basic building block for our final `PulseGate` type, which we will use to model SCE circuits; however, currently it is only able to reflect the I/O behavior of a gate, e.g., the `TimedPulse` information. In reality, each SCE gate additionally keeps track of unique, internal state. For example, the LA and FA gates each keep track of their own distinct information on different transition delays and timing constraints. We model this information with the type `StateVector` in Listing 4. The `StateVector` keeps track of all internal state for a given SCE element. For the most part, we use the definitions provided in the PyLSE paper, making small simplifications. The PyLSE Machine needs to keep track of roughly the following state: (1) current state; (2) each input's last arrival timestamp; (3) time required to transition from one state to another given an input; (4) time taken for an output to be observable; (5) timestamp of the end of the unstable period; (6) various timing constraints related to hold/setup time.

Now that we have the type of internal state represented, we must decide how best to incorporate it into the basic `MealyMachine` type. We *could* expose a gate's internal state by lifting the state into its type signature as we showed for `MealyMachine`, i.e., $I \to S \to O \times S$ where S is the type of the state, but then, as a practical matter, it becomes difficult to write generic composition functions. That is, gates should only be composable in terms of their I/O behavior (i.e., the pulses sent and received)—each gate's internal state should be hidden from the other gates they are composed with. We need an individual gate's state to be encapsulated so that it does not appear in that gate's

```
record StateVector (X : Set) (n : ℕ) : Set where
  field
    -- current state
    qcurr : X
    -- when did i last see this input?
    Ω : Vec Pulse n → ℕ
    -- from a given state and input, how long does it take to update?
    tmap : X → Vec Pulse n → ℕ
    -- tarr + ttrans
    tdone : Time
    -- same as tmap but for outputting
    firing-delay : Time
    -- setup time constraints (list of deadlines for given pulses)
    θ : (X × (Vec Pulse n)) → List ((Vec Pulse 2) × Time)
```

Listing 4. The StateVector data structure used to hold timing and node information. The list θ contains the *setup time* constraints which are timestamps beyond which receiving inputs would result in a violation.

```
initDataC : StateVector S₃ 2
qcurr initDataC = s₃0
Ω initDataC = λ x → 0
tmap initDataC = λ x v → 0
tdone initDataC = 0
firing-delay initDataC = 8
θ initDataC = λ (x,y) → []
```

Listing 5. The data that defines and initializes the LA gate. The type S_3 is used to represent unique internal state of the LA gate.

type-level signature. When the overall circuit (the composition of gates) is stepped forward in time, each gate's state should be automatically (and invisibly) looped back to itself to advance and maintain their individual states as appropriate. The `MealyMachine` type does not fit this requirement (because it exposes the state information), and so we need to extend our definitions further to get a suitable PulseGate definition.

To implement the desired behavior we turn to a concept explored in functional reactive programming, namely that of a *loop*, or *trace* [Hasegawa 1999]. Several structures in functional programming permit the construction of a *loop* operator. The basic type signature takes a function of type $I \times S \longrightarrow O \times S$ and turns it into one of $I \longrightarrow O$, that is, it invisibly feeds back the S parameter yielding a function that only exposes its I/O parameters. The *loop* operator for `PulseGates` can be implemented as below in Listing 6, where the type S is the type of internal state being "fed back". Note that in the code development, the `PulseGate` type is identical to the above Mealy Machine type, simply for ease of type signatures; in practice, every SCE circuit is indeed the result of a `loop` operator with the above StateVector feedback.

We briefly call attention to the two equations that define the `loop` function. Contrary to typical Haskell-style definitions, the record fields of the PulseGate are called on the function itself. This is a corecursive function as opposed to a recursive one. The interpretation is that given some term `loop fdbk m` and an input `tp` (a `TimedPulse`), calling the `resp` function on this structure should yield the right-hand side of the equation. The function is defined corecursively, that is to say, it is defined by what the destructors of the record do to terms of the given type. With these definitions in place we can describe the `PulseGate` data structure.

```
loop : (n1 n2 : ℕ) (fdbk : S) →
      PulseGate (TimedPulse n1 × S) (TimedPulse n2 × S) →
      PulseGate (TimedPulse n1) (TimedPulse n2)
resp (loop fdbk m) tp = proj₁ (resp m (tp , fdbk))
step (loop fdbk m) tp = loop (proj₂ (resp m (tp , fdbk)))
                             (step m (tp , fdbk))
```

Listing 6. The *trace* operator implemented for the PulseGate data structure.

Definition 1 (PulseGate)**.** The `PulseGate` is a looped `MealyMachine` with I/O given as `TimedPulses` and feedback information corresponding to `StateVectors` representing a specific SCE gate's unique internal state.

To put Definition 1 into practice, we construct an example corresponding to the Last Arrival gate from Figure 1. We proceed in two steps. In the first step we create a function, the underlying transducer, that exposes the internal state information, and use it to form a `MealyMachine`. In the second step we apply the `loop` operator to this machine and its associated initial data to obtain the full circuit. The underlying transducer of the Last Arrival gate is pictured below, in abbreviated form in Listing 7. It defines how the gate behaves according to pulse inputs and how internal state, such as its own clock and timing constraints, are updated.

```
LA-transducer : TimedPulse 2 × StateVector S₃ 2 →
                TimedPulse 1 × StateVector S₃ 2
LA-transducer ((v,tarr), ⟨qcurr, Ω, tmap, tdone, fd, θ⟩)
                with (validInput tarr tdone)
LA-transducer (v@(*::*::[], tarr), ⟨s₃0, Ω, tmap, tdone, fd, θ⟩) | true =
  ((*::[]), (tarr + fd)), ⟨ s₃0 , (λ s → tarr), tmap, (tarr + fd), fd, θ⟩
...
LA-transducer (v@(*::∅::[], tarr), ⟨ s₃0, Ω , tmap, tdone , fd , θ ⟩) | true =
  ((∅::[]), (tarr + tmap s₃0 (proj₁ v))), ⟨s₃0 , (λ s → tarr), tmap, (tarr +
       tmap s₃0 (proj₁ v)), fd, θ⟩
...
```

Listing 7. A few example lines from the definition of the transducer function for the LA gate. We can see the last equation gives the case where the input holds a pulse on the first wire but not the second and the element is in the $s_3 0$ case. The `true` guard establishes the pulse's arrival time does not trigger an error. The first component of the right hand side of the equation shows that the element does not pulse while the second component takes care of state bookkeeping, for instance, updating the most recent arrival time on this wire.

The function pattern matches on the current state *qcurr* (in the sense of a node in the PyLSE Machine graph) and the input slices, which can consist of a pulse on each wire, a pulse on no wires, or pulses on one or the other wires. There can also be error signals ingested which simply propagate through the circuit as a whole. We follow PyLSE in that we assume pulses are tagged with an absolute timestamp, as opposed to a relative timestamp. The StateVector then keeps track of the timestamps of each I/O pulse, as well as when the element is ready to accept new inputs. The syntax "`with (validInput tarr tdone)`" conveys that we should only begin computing the transition if the given input does not conflict with the element in any way. The function *validInput* is based on the PyLSE paper's *ERROR-K-TRAN* and *ERROR-K-CONS* rules; intuitively it sends to an error state (and emits an error signal) if the timing constraints are violated in some way. In all then, the way to interpret the `LA-transducer` function is to look at what kind of `TimedPulse` arrived,

what internal state the transducer was holding onto, compute the next state, whether or not there is an output pulse, and then the times taken for transitioning or firing and when the next stable period begins.

We can now proceed to the second step which combines the underlying transducer with its initial state information under the `loop` operator and is pictured in Listing 8. This data structure satisfies all of our main requirements: At the type level, it exposes only the I/O behavior of the circuit, allowing it to be more easily composed; it keeps track of unique internal state via the underlying transducer function; and it runs infinitely, always able to respond to new inputs (modulo error-inducing inputs).

```
-- the Last Arrival gate
LA : PulseGate (TimedPulse 2) (TimedPulse 1)
LA = loop initDataC (arr LA-transducer)
```

Listing 8. The full Last Arrival gate as represented in the Citrus language. The `loop` The `arr` operator lifts the underlying transducer function into a `PulseGate`.

3.3 Composition Operators

Functional arrows benefit from several composition operators that allow one to construct large dataflow-like graphs. We extend our `PulseGate` structures with a handful of such composition operators. We have defined operators to put PulseGates in parallel, in sequence, and to combine a pair of PulseGates but propagate only the behavior of the first; together these operators are sufficient to define a wide array of possible compositional behaviors and give the user ample capability to design SCE circuits as if they were dataflow-like networks. This design improves upon PyLSE itself in that the composition information is represented at the type level, which facilitates easier debugging as well as easier programming. Below we describe the sequence and parallel composition operators as examples. The sequence operator (Listing 9) funnels the output of one PulseGate as the input to the next. The parallel composition operator (Listing 10) is capable of processing inputs simultaneously or, if inputs come only on a single wire, one at a time as they are ingressed.

```
_>>>_ : (PulseGate I M) → (PulseGate M O) → PulseGate I O
resp (m₁ >>> m₂) = resp m₂ ∘ resp m₁
step (m₁ >>> m₂) = λ x → (step m₁ x) >>> (step m₂ (resp m₁ x))
```

Listing 9. The sequence operator chains two PulseGates one after the other.

```
_***_ : (PulseGate (TimedPulse i) (TimedPulse j)) →
        (PulseGate (TimedPulse m) (TimedPulse p)) →
        PulseGate (TimedPulse (i + m)) (TimedPulse (j + p))
resp (_***_ m₁ m₂) = λ vim → resp m₁ (proj₁ (splitAtTP i vim))
       ++TP resp m₂ (proj₂ (splitAtTP i vim))
step (_***_ m₁ m₂) = λ vim → step m₁ (proj₁ (splitAtTP i vim)) ***
      step m₂ (proj₂ (splitAtTP i vim))
```

Listing 10. The parallel composition operator. The indices *i,j,m,p* are natural numbers indicating widths.

The composition operators do not impose any timing penalties beyond those imposed by the gates being composed, that is, because the hardware itself switches near instantaneously, we do

not add delays or added elapsed time to the composition operators themselves. Finally, we note that networking circuits together frequently requires operations like duplicating and forking the wires. As an example, we present the "cloning" function below which duplicates one of its input wires in a specific order. In Citrus these "wire management" functions are turned into PulseGates and then composed using the same composition operators.

```
cloning : TimedPulse 3 ⟶ TimedPulse 4
cloning (a :: b :: c :: [] , t) = (a :: b :: a :: c :: [] , t)
```

Listing 11. A simple function which duplicates its input in a specific pattern. Functions like these are necessary to do so-called "wire-management" in larger designs.

3.4 Programming with PulseGates

In this section we present a basic example of programming with the Citrus framework. We use as our running example the `min-max` element, or comparator. It is visualized in Figure 4 and displayed as Citrus code in Listing 12. The comparator takes two inputs and pulses its first wire on the earliest input pulse, and its second wire on the latest input pulse. This has the effect of sorting the inputs from earliest to latest. This element is implemented as a composition of three more basic elements, the *S* gate, or splitter (which duplicates a wire), and the familiar *FA* and *LA* gates. We can see the close correspondence between the visual description of the SCE circuit and the Citrus version as code.

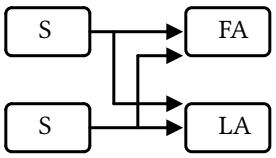

Fig. 4. The comparator visualized as a network of SCE elements.

```
min-max : PulseGate (TimedPulse 2)
                    (TimedPulse 2)
min-max = (S *** S) >>>
          ((arr perm0) >>> (FA *** LA))
```

Listing 12. The comparator element as a term in Citrus. `perm0` is a wire management function and `arr` promotes it to a PulseGate.

Finally, we give some examples of running simulations of the above circuits. We first define a few sample inputs shown in Listing 13. For instance, the first input is on the second wire. We submit these inputs to the comparator circuit and observe the outputs, as shown in Listing 14.

We need to submit both inputs to the comparator to fully realize its behavior. We can see from `out` that the behavior of the comparator correctly pulses its first wire at the input timestamp plus its internal transition time/firing delay. During this period, receiving another input would be an error, as shown by result `out'`. However, the third input `test-pulse-3` is sufficiently far in time and can trigger the comparator completely. That is, the comparator pulses its second wire appropriately delayed in time. This example demonstrates that Citrus allows SCE developers to simulate the semantics of SCE circuits for testing and prototyping in a simple, functional style.

3.5 Limitations and Future Work

We discuss two limitations in Citrus, a comparatively simpler semantics and abstractions from hardware complexity.

```
-- a pulse on the second wire at time 4
test-pulse-1 : TimedPulse 2
test-pulse-1 = (∅ :: * :: []) , 4

-- a pulse on the first wire one time point later
test-pulse-2 : TimedPulse 2
test-pulse-2 = (* :: ∅ :: []) , 5

-- a pulse on the first wire 8 time units later
test-pulse-3 : TimedPulse 2
test-pulse-3 = (* :: ∅ :: []) , 13
```

Listing 13. We establish test inputs as TimedPulses. These are pairs of a vector and a time value where , in Agda, a comma separates the elements in a pair and :: is the cons operator.

```
-- * :: ∅ :: [] , 13
out = proj₁ ((runPG min-max) test-pulse-1)

-- this steps the circuit forward one time step
nxt = proj₂ ((runPG min-max) test-pulse-1)

-- this produces an error (err :: * :: [] , 10)
out' = proj₁ ((runPG nxt) test-pulse-2)

-- ∅ :: * :: [] , 18
out'' = proj₁ ((runPG nxt) test-pulse-3)
```

Listing 14. Test inputs are simple TimedPulse objects or vectors of pulses/absences/errors at specific timestamps. Running a circuit against a sequence of test inputs involves the helper function runPG. We show in comments the output of running the functions.

Compared to PyLSE, Citrus simplifies the semantics in two ways. In the first, while Citrus can represent more complicated timing constraints, such as setup time, it currently only takes into account basic timing and firing delays when running computations. Extra timing constraints, such as setup and hold time, are additional checks on the arrival times of pulses based on what internal state an element is in. In the second, Citrus, by encapsulating state to each element and composing these as functions, considers fewer possible interleavings of pulse communication between elements in the circuit. That is, by using a network of timed automata in parallel (which is used to generate one very large timed automaton), PyLSE considers a much larger set of possible execution traces. In the first simplification, we believe the additional timing constraints can be acted on straightforwardly, in a manner similar to the current check on `validInputs`. The second simplification is more interesting, as the PyLSE language, despite representing more possible traces, strives to rule them out by invalidating any circuit that does not conform to the intended "straight-line" behavior. A correct circuit in PyLSE then is ultimately a discrete sequence of state changes. We claim this is roughly analogous to the operation of Citrus; however, we are actively working to relax the restrictions on execution traces inherent in Citrus. In total, we do not think these issues detract from the basic functionality expected of a model of SCE and moreover that the simplifications allow Citrus to reveal several interesting algebraic facts about SCE.

The second limitation of Citrus is with respect to the hardware complexity of the material physics and electronics of SCE. For instance, we do not represent thermal noise or jitter; however, the

underlying state-management issues we abstract in Citrus are, we believe, inherent to having information stored in the form of magnetic flux quanta and then transferred and manipulated in the form of single flux quantum (SFQ) voltage pulses. Path balancing, dual-clocking, and other "digital equivalent" techniques, will not change the abstractions we develop here and the proofs of correctness of fundamental circuit transformations should apply equally. There are certainly unique SCE device-level failure modes outside of Citrus domain of abstraction (e.g. faults due to Abrikosov vortex pinning) in the same way there are device level failure modes for CMOS below the level of RTL (e.g. hot carrier injection).

4 Equational Reasoning

Now that we have a formal representation of `PulseGates` and of SCE circuits as compositions of PulseGates (i.e., as PyLSE Machines), we turn our attention to the notion of equivalence between SCE circuits. That is, when are two PyLSE Machines behaviorally equivalent, or in other words, when do two machines yield the same observable outputs given the same inputs? To this end, we adapt the notion of bisimulation and discuss how it may be used to prove the equivalence of two PyLSE Machines in Section 4.1. We discuss two finer points related to these notions in Section 4.2 which are required to fully complete our proof technique.

4.1 Equivalence between PulseGates

The classical treatment of equivalence for reactive systems is in terms of bisimulation over infinite labeled transitions systems (LTS) [Aceto et al. 2007]. Because the LTS do not have a termination condition the equivalence between them cannot depend on recursing to a base case (which computes a least fixpoint); instead it must depend on the greatest fixpoint [Sangiorgi 2011]. To establish bisimilarity in this framework we check: (1) that if one LTS can make a transition α then the other LTS can make a transition α; and (2) the resulting states of the two LTS are bisimilar. Mathematically, this is a coinductive relation, and its coinductive nature is captured clearly by the `PulseGate` coinductive records that we use to describe SCE circuits.

The framework of bisimulation has been extended to cover time-sensitive LTS, such as timed automata. There are several different notions of timed bisimulation [Aceto et al. 2007], including "timed bisimulation" and "time-abstract bisimulation". The former requires that the two LTS must make the same transitions with the same delays, which is too restrictive for our purposes (e.g., proving that one SCE circuit is an optimized version of a second, with equivalent behavior but smaller delay). Time-abstract bisimulation, however, turns out to be a good choice to prove circuit equivalence. In this setting the two LTS must take the same transitions but are allowed to do so with differing delays. Intuitively, it says that for two SCE circuits to be time-abstract bisimilar there must be some amount of time they each delay (which can be different for each one) such that, given the same input sequence, they must respond with the same output sequence. Note that time-abstract bisimilarity subsumes weak bisimilarity (which allows an LTS to take "silent" transitions) because a silent transition is essentially equivalent to an additional delay.

This notion of bisimilarity works well for SCE circuits because SCE circuit semantics depend on the timing of pulses *relative* to the circuit characteristics such as transition and firing delays, rather than absolute timing. Time-abstract bisimilar circuits thus have the same "cycling" behavior—that is, if we use the Mealy Machines in Figure 1 as a convenient reference, an input sequence that causes one circuit to go from its *Init* state through either the $a - recv$ or $mathitb - recv$ states and then back to *Init* (respecting its various time delays) will also cause another bisimilar circuit to do the same (respecting *its* various time delays).

We begin with bisimilarity as it is presented in Abel's work on formal languages [Abel 2016] which we reproduce in Listing 15, specialized to `PulseGates`. This type is a coinductive record

called _ ≈ ⟨_⟩ ≈ _, where the first and last underscores stand for the circuits being compared and the middle underscore stands for the depth of observation, e.g., $A \approx\langle\infty\rangle\approx B$ means that A and B are time-abstract bisimilar for arbitrary depths of observation. We direct the reader to the `eqr.agda` file in the supplementary material for the full development; here we show a simplified presentation in Listing 15 removing the size annotations for readability. The record takes two PulseGates as inputs and has two fields, ≈`resp` and ≈`step`. The first, ≈`resp`, represents a proof that the outputs of the two gates are equivalent; the second, ≈`step`, establishes that the stepped gates (yielded as a result of a PulseGate `step`) also bear the equivalence relationship. In more traditional terminology, these two fields state that the heads of the two gates are equal and the tails are (time-abstract) bisimilar.

```
record ≈⟨ ⟩≈ {A B : Set} (m₁ m₂: PulseGate A B) : Set where
  coinductive
  field
    ≈ν : resp m₁ ≡ resp m₂
    ≈δ : ∀ (a : A) → (step m₁ a) ≈⟨ ⟩≈ (step m₂ a)
```

Listing 15. The bisimilarity relation follows the definition pattern we used for PulseGates. A proof that two PulseGates are equivalent requires a proof that all inputs are equal (this is the ≡ symbol) for arbitrary delays and that for all inputs after arbitrarily delaying the continuations of the gates are also equivalent.

The relation in Listing 15 is an equivalence relation, satisfying the properties of reflexivity, symmetry, and transitivity. Proofs of these facts are straightforward and based on simple copattern matching. However, this current definition is incomplete because it does not incorporate any notion of *time*. In order to address this issue we must explain how we can reason about sequences of pulses over time.

4.2 Ordered Sequences and the Coinduction Hypothesis

There are two important issues that remain to be solved before we can prove two SCE circuits as equivalent. The first problem is that we need to establish that for all possible sequences of inputs (recalling that an input is a vector of pulses, one per wire, plus a timestamp), two circuits output the same thing; however, "all possible inputs" is too many to feasibly check.

The stateful semantics of SCE gates provides us with an answer: an SCE gate is assumed to start in some initial state, then to get at least one pulse on each wire (cycling through some number of states internal to that gate) before returning back to the initial state. For example, the Last Arrival gate and First Arrival gate from Figure 1 both have two input wires (represented as possible inputs `a` or `b`) and, once having received a pulse on some wire, cannot return to their initial states before receiving an input on the remaining wire; the difference is that Last Arrival only fires when it receives a pulse on the remaining wire that hasn't seen a pulse yet, while First Arrival fires immediately upon receiving the first pulse. Let us call the period from when a gate is in its initial state and receives some pulse to the point it has received a pulse on all input wires and returned to its initial state a *regime*. Then we only need to check sequences of inputs that advance through a regime, completing a gate's cycle of behavior. In particular, we can ignore the possibility of any inputs that advance a gate's PyLSE Machine along a reflexive edge with no output, because that input has no effect on the gate's behavior—for example, if a Last Arrival gate is in state $a - recv$ and receives another pulse on the `a` gate, that input has no effect and we don't need to consider it when establishing equivalence. We summarize the situation as follows.

- **Repeated pulses.** It's possible for multiple pulses to occur on a single wire without any intervening pulses on the other wires. For all the gates we consider, the setup time constraints, θ, are empty and so we can be confident that a repeated pulse leaves Circuit1 and Circuit2 in the same configuration as before the repeated pulse, and hence we can ignore this possibility when enumerating cases. There is an issue with timing, as repeated pulses will in general update the internal clocks of SCE gates. We treat this issue in more detail below.

- **Simultaneous pulses.** Simultaneous pulses on two wires could occur in theory (though no known SCE gates allow this in reality). The behavior of simultaneous pulses on wires x and y would be either the behavior of receiving x then y or the reverse; which one is unimportant since we cover both of these cases in the proof. The timing component of this case poses no problem as the circuit essentially bypasses the "dirty" state and returns to its "clean" initial state immediately. That is, there is no risk of getting stuck in an internal state and then violating a timing constraint.

As mentioned in the case of repeated pulses, we must expand on the timing implications of this approach to proving equivalence. We are interested in two SCE gates simulating each other's I/O behavior up to certain time delays. We are satisfied, then, if one circuit delays for awhile and then mimics the other circuit's behavior; in fact, this could be a sign that one circuit is some optimized version of the other. Nevertheless, it could be that a circuit delays "too much" and either one machine risks receiving a pulse that violates an internal timing constraint, or the two circuits somehow fall out of (delayed) lock step. The approach we take to remedying this problem is to consider only a representative sequence of input pulses. We know in advance when a sequence of timed pulses will trigger a timing violation. In fact, we can compute this information *a priori* from the initial data or we can interactively determine it by running a circuit against inputs and checking the outputs. The first approach lends itself to automation and while we do not provide that facility in this paper, we do leave it to future work. Given a sequence of inputs that are correctly timed, we know the gate will conform to its intended functionality. It is sufficient then to test circuits for equivalence up to these "well-timed" sequences. To make this clearer, we demonstrate in Listing 16 and Listing 17 the two concepts that underly the above discussion of time. The first shows how we represent and feed input sequences to `PulseGates` while the second describes the "cyclic" property of SCE gates according to a coinduction hypothesis.

```
let (FA-out-1 , FA-next-1) = runPG FA (inp-A t1)
  (FA-out-2 , FA-next-2) = runPG FAcomm-next-1 (inp-B t2)
  (FAcomm-out-1, FAcomm-next-1) = runPG FA (flip (inp-A t1))
  (FAcomm-out-2, FAcomm-next-2) = runPG FAcomm-next-1 (flip (inp-B t2))
in (FA-out-1 ~ FAcomm-out-1) ×
  (FA-out-2 ~ FAcomm-out-2) ×
  (FA-next-2 ≈⟨ ⟩≈ FAcomm-next-2)
```

Listing 16. We specify inputs by *sequences* of pulses for instance `inp-A` and `inp-B`. We then run the circuits on these inputs and keep track of the inputs and outputs.

By collecting the distinct input sequences together we are able to run the machine through the sequences to determine their complete profile of behavior. It is the outputs along the way that need to be equivalent and the stepped machines that must be bisimilar. This proof setup brings up the last remaining problem to actually proving two circuits bisimilar: The PulseGates are coinductive

```
coind : proj₂ (runPG (proj₂ (runPG FA (inp-A t1))) (inp-B t2))
≈⟨ ⟩≈
  proj₂ (runPG
    (proj₂ (runPG FA
      (∅ :: * :: [] , proj₂ (inp-A t1))))
      (* :: ∅ :: [] , proj₂ (inp-B t2)))
```

Listing 17. A coinduction hypothesis is required to validate the basic behavior of all the SCE gates. This hypothesis corresponds to the cyclic behavior that is unique to all SCE gates.

and able to be "stepped" forward infinitely, indeed modeling the real life SCE gates; but when can we safely stop for the purposes of our proof? We take advantage of the built-in notion of "cyclicity" that guarantees well-designed SCE circuits will start in some initial, ready state, receive inputs, perform some work, and then eventually return to the initial state. This cyclicity is in reality what we are capturing with our coinductive structures, not solely the ability to be stepped forward infinitely.

To put this in perspective, we can see in Listing 16 that after supplying both inputs to the FA gate, we could simply keep doing this, stepping the circuit repeatedly with new inputs. This would be strange however, because we know that the function of the FA gate is to return to its initial state, having performed its output, at the completion of this sequence. We would like to somehow wrap up this observation and use it to complete the proof. In the setting of coinductive proofs, this type of maneuver is called a "coinductive hypothesis", dual to an inductive hypothesis. The coinductive hypothesis captures a cycle of behavior that appears in the otherwise infinite stepping of a coinductive object. Thus, in the Citrus language, every basic SCE gate comes with a coinductive hypothesis (derived directly from the intended behavior of that gate) establishing the correctness of the behavior of the circuit. Put differently, we are able to build SCE gates in a "correct by construction" fashion. An example of the coinductive hypothesis corresponding to the FA gate above and the given input sequence is shown in Listing 17. With all of this in place we are able to more formally state the definition of `PulseGate` equivalence. These are guidelines for the proof technique, governing the examples, and note present as types in the code. We begin with the concept of the ordered sequence that does not trigger timing violations.

Definition 2 (Well-timed Sequence)**.** For all $i\ j : \mathbb{N}$, let s be a sequence of type (`TimedPulse` i) and m a `PulseGate` (`PulseGate` i) (`TimedPulse` j), then s is well-timed if for all timestamps in the sequence s, (`resp m s`) does not trigger an error signal.

This definition expresses that, while applying an input to a `PulseGate`, the outputs generated (by feeding the output into the `step` function to continue the circuit) do not trigger an error signal. The definition captures the intuition that there is a schedule under which input pulses should occur of a given SCE element to function correctly. Essentially, the inputs should be separated in time sufficiently for the circuit to make all of its internal state changes without interruption. This is essentially the approach taken in [Christensen et al. 2022] which constructs a priori a sequence of traces at specific times and calculating the correct output times. Citrus implements this technique with a strategy similar to that of the coinductive stack given in [Abel 2016], namely by using `let...in` statements to manually schedule the input sequence and then copattern match on the subsequent results.

There are two minor issues with this approach. The first is that we determine well-timed sequences manually. The second is that we depend on a representative trace while a circuit may have many correct execution traces. Determining well-timed sequences should be automatable and we believe this to be straightforward. On the other hand, the use of a representative sequence

versus a set of sequences is subtler issue. We speculate the two should be interderivable and are working on this issue actively. Nevertheless, in practice, given a representative sequence, we are able to proceed with proving two circuits functionally equivalent. Indeed, with the definition of well-timed sequences and bisimilarity as a coinductive relation, we present the definition of `PulseGate` equivalence.

Definition 3 (`PulseGate` Equivalence)**.** For all $i\ \ j\ :\ \mathbb{N}$, m_1, m_2 : `PulseGate` (`TimedPulse` i) (`TimedPulse` j), (inp:`TimedPulse` i), m_1 is equivalent to m_2 if there is a well-timed sequence s and time delays d_1, d_2, such that for all intermediate outputs (until the coinduction hypothesis is reached):

(1) (`resp` m_1 s) ≡ (`resp` m_2 s)
(2) $\texttt{step}_k(m_1) \approx\langle\ \rangle\approx \texttt{step}_k(m_2)$

where $\texttt{step}_k$ represents the number of `steps` that must be applied to the circuit m before it can invoke its coinduction hypothesis and return to its clean state.

Having defined `PulseGate` equivalence we are able to set on a firm foundation many equational laws which SCE circuits obey but which have yet to be proven formally. We pursue two sets of case studies along these lines in Section 5.

5 Case Studies

We demonstrate the usefulness of Citrus by using it in a set of case studies. The first set of case studies (Section 5.1–Section 5.3) examines several "laws" about SCE gate composition that depend on equivalences that SCE designers believe should be correct, but that have never formally been proven. The last case study (Section 5.4) examines a non-trivial SCE circuit taken from the PyLSE paper [Christensen et al. 2022], a bitonic sorter, and proves a property about it which PyLSE is unable to prove via model checking (i.e., by translating the PyLSE Machine to a timed automata and using a third-party model checker). All of these proofs have been mechanized and are available in the supplementary material.

5.1 Temporal Invariance

The first SCE composition law states that delaying two inputs by k time units and running them into a First Arrival gate is the same as running two inputs into a First Arrival and then delaying by k time units. Stated algebraically, where D is a delay and FA is a First Arrival gate:

Proposition 5.1.1 $(\forall(a\ \ b : TimedPulse\ 2)(k : Time),\ \ FA(D_k(a), D_k(b)) = D_k(FA(a, b)))$**.**

We can represent the circuits on each side of the equality in Citrus as shown in Listing 18 using the combinators defined earlier in the paper:

```
Delay-FA = ((Delay {1} k) *** (Delay {k} {1} k)) >>> FA
FA-Delay k = FA >>> Delay k
```

Listing 18. The two circuits used in Proposition 5.1.1

To prove the equivalence we proceed in three steps. The first step is to establish a coinduction hypothesis stating when the gate may be considered "done" and able to return to its initial state. The second step is to construct the sequence of inputs, keeping track of the intermediate outputs and updated circuits. The third step is to write down the proof obligations, namely, that the intermediate outputs are equal and that the final, updated circuits are bisimilar. We refer to this pattern as "setting

```
temp-inv :
-- the coinduction hypothesis
(coind : proj₂ (runPG (proj₂ (runPG Delay-FA (inp-1 0))) (inp-2 11))
        ≈⟨ ⟩≈
        proj₂ (runPG (proj₂ (runPG FA-Delay (inp-1 0))) (inp-2 11))) →
-- constructing the sequence of inputs
let (DelFA-out-1 , DelFA-next-1) = runPG Delay-FA (inp-1 0)
    (DelFA-out-2 , DelFA-next-2) = runPG DelFA-next-1 (inp-2 11)
    (FADel-out-1 , FADel-next-1) = runPG FA-Delay (inp-1 0)
    (FADel-out-2 , FADel-next-2) = runPG FADel-next-1 (inp-2 11)
-- the proof obligations
in (DelFA-out-1 ≡ FADel-out-1) ×
   (DelFA-out-2 ≡ FADel-out-2) ×
   (DelFA-next-2 ≈⟨ ⟩≈ FADel-next-2)
```

Listing 19. The proof setup for Proposition 5.1.1. It consists of three parts: assuming the coinduction hypothesis; the relevant input sequence; and the equalities and equivalences that form the proof obligations.

up the equivalence" or "proof setup" and make use of it in the remaining case studies. Listing 19 shows the proof setup for the proof of Listing 18.
With this setup we can then prove the theorem (i.e., the three proof obligations given in Listing 19) with three corresponding equations as shown in Listing 20:

```
proj₁ (temp-inv coind) = refl
proj₁ (proj₂ (temp-inv coind)) = refl
proj₂ (proj₂ (temp-inv coind)) = coind
```

Listing 20. The proof of temporal invariance consists solely of three equations which Agda validates as true.

Using Citrus with all of its provided definitions and operators, there is minimal difficulty in setting up and proving the above theorem. The only manual work it requires is to establish a given temporal path, e.g., the timestamps required for the input pulses, and the corresponding coinduction principle. It might be possible to automate this manual work as well using arithmetic solvers, but we leave it for future work.

5.2 Commutativity of the FA and LA Gates

The second SCE composition law states that both the First Arrival and the Last Arrival gates are each commutative, i.e., the behavior of both gates is the same regardless on which wire the input pulses arrive. Proposition 5.2.1 states the case only for the First Arrival gate; the statement is nearly identical in the case of the Last Arrival gate.

Proposition 5.2.1 $(\forall (a \;\; b : TimedPulse\ 2), \;\; FA(a, b) = FA(b, a))$**.**

The proof setup follows the same pattern as that of Proposition 5.1.1 with one exception: In this case we do not need a coinduction hypothesis as the final proof obligation is merely reflexively true. Agda is able to validate this on its own (through the use of a helper function). Listing 21 shows the setup code:

This proof is even simpler than the proof of temporal invariance, as we do not need a coinduction hypothesis, because the gates have identical internal state as determined by Agda. Thus we can resolve the proof obligation using the helper function `≈refl` which discharges reflexively bisimilar

```
FA-comm :
  let (FA-out-1 , FA-next-1) = runPG FA (inp-1 0)
      (FA-out-2 , FA-next-2) = runPG FA-next-1 (inp-2 11)
      (FAcomm-out-1 , FAcomm-next-1) = runPG FA (flip (inp-1 0))
      (FAcomm-out-2 , FAcomm-next-2) = runPG FAcomm-next-1 (flip (inp-2 11))
  in (FA-out-1 ≡ FAcomm-out-1) ×
     (FA-out-2 ≡ FAcomm-out-2) ×
     (FA-next-2 ≈⟨ ⟩≈ FAcomm-next-2)
proj₁ (FA-comm) = refl
proj₂ (proj₂ (FA-comm)) = refl
proj₂ (proj₂ (FA-comm)) = ≈refl
```

Listing 21. The First Arrival Gate is commutative in the sense that its inputs can be given in any order.

terms (analogous to `refl` for inductive terms). The proof for Last Arrival is identical so we do not show it here.

5.3 Distributivity of the FA and LA Gates

The third SCE composition law states that First Arrival distributes over Last Arrival. This law is useful practically, in that it permits us to replace a circuit with four gates by a circuit with three gates. However, it is also interesting from a logical perspective, because the classical analogues of First Arrival and Last Arrival are the OR and AND gates. In Boolean logic conjunction and disjunction enjoy a distributive relationship: $(M \vee (A \wedge B)) = (A \vee M) \wedge (A \vee B)$. Here, we show that the SCE analogue to this distributive law holds as well.

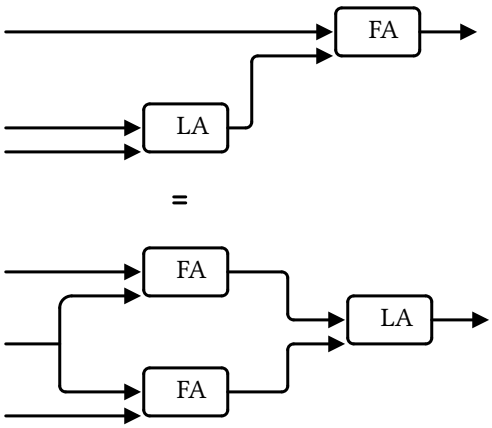

Fig. 5. Schematic representation of the distributive law translated to pulse-based SCE gates. We refer to the top circuit as Circuit1 and the bottom circuit as Circuit2.

This case study comes from common transformations done to classical CMOS circuits to optimize area and performance. The question is whether Circuit1 $\approx\langle\,\rangle\approx$ Circuit2 (where *Circuit1* and *Circuit2* are the top and bottom circuits in Figure 5, respectively); if so then a designer can optimize a circuit that uses *Circuit2* (which has more gates and a longer delay) by replacing it with *Circuit1* (which has fewer gates and a shorter delay). SCE designers strongly suspect the truth of this equivalence, but there is no proof that it is correct.

```
Circ1-eq-Circ2-MAB :
  (coind : proj₂
    (runPG (proj₂ (runPG (proj₂ (runPG Circ1 inp-M')) inp-A')) inp-B')
    ≈⟨ ⟩≈
    proj₂ (runPG (proj₂ (runPG (proj₂ (runPG Circ2 inp-M)) inp-A)) inp-B)) →
let (circ1-out-1 , circ1-next-1) = runPG Circ1 inp-M'
     ...
     (circ2-out-1 , circ2-next-1) = runPG Circ2 inp-M
     ...
  in (circ1-out-1 ≡ circ2-out-1) ×
     ...
    (circ1-next-3 ≈⟨ ⟩≈ circ2-next-3)
proj₁ (Circ1-eq-Circ2-MAB coind) = refl
proj₁ (proj₂ (Circ1-eq-Circ2-MAB coind)) = refl
proj₁ (proj₂ (proj₂ (Circ1-eq-Circ2-MAB coind))) = refl
proj₂ (proj₂ (proj₂ (Circ1-eq-Circ2-MAB coind))) = coind
```

Listing 22. Distributive law proof setup. Given a sequence of inputs *M,A,B* there are inputs *M',A',B'* such that the two are behaviorally equivalent.

The distributive law we are trying to prove may be stated as Circuit1 ≈⟨ ⟩≈ Circuit2. The proof setup is shown in Listing 22. In order to decompose the problem, we will investigate it by cases. As described earlier the SCE gates work in a cyclical fashion, returning to an initial configuration after receiving a pulse on every wire. This behavior means that there are eight cases to consider, one for each possible ordering of pulses arriving on the *A*, *B*, and *M* wires. For example, in one case we examine the input sequence in which the *M* wire pulses, then the *A* wire followed by the *B* wire. We prove that the outputs of the two circuits Circuit1 and Circuit2 for this input sequence are equal. At this point, the gates should each be back to their respective initial configurations—except that they are not exactly. In an untimed setting this would be true, but in a timed setting they have each updated their clocks and so are technically not the same as the original configurations (though they are the same up to that timing information). To overcome this problem we employ a standard coinductive technique analogous to strengthening an inductive hypothesis: we assume a coinductive hypothesis stating that the new configurations are time-abstract bisimilar. Essentially, this hypothesis states that we went through one step of testing that the circuits yield the same behavior for the same input sequence, and that the remaining infinite steps will look exactly the same—thus if the first step worked (which it did) then the remaining steps must work as well.

5.4 Bitonic Correctness

Our final case study is on bitonic sorters, a real-world SCE circuit that was also used as a case study in the Pylse paper [Christensen et al. 2022]. A bitonic sorter is a parallel sorting network composed of min-max blocks. An *n*-width bitonic sorter implementation is correct if it pulses its *n*th output wire on the *n*th input to arrive. PyLSE is unable to prove the correctness of a particular 4-bit SCE bitonic sorter implementation via model-checking, because the compiled timed automaton to be model checked contained too many states to be efficiently validated. However, with Citrus its proof is only a few lines long. We show the Citrus program implementing the bitonic sorter in Listing 23 and its proof of correctness in Listing 24.

The proof in Listing 24 is establishing the four proof obligations (in addition to the fifth obligation establishing bisimilarity), i.e., that after running all input pulses, the sequence of output pulses should come in order, from first to last wire of the bitonic sorter itself. Given the machinery provided

```
bitonic-sorter-4 : PulseGate (TimedPulse 4) (TimedPulse 4)
bitonic-sorter-4 = (min-max *** min-max) >>> ((arr perm1) >>>
                   ((min-max *** min-max) >>> ((arr perm2) >>>
                   ((min-max *** min-max)))))
```

Listing 23. The definition of a 4-bit bitonic sorter. The *perm*1 and *perm*2 functions are wire management arrows.

```
bitonic-correct :
(coind : proj₂ (runPG (proj₂ (runPG (proj₂ (runPG
(proj₂ (runPG bitonic-sorter-4 test-pulse-a)) test-pulse-b)) test-pulse-c))
    test-pulse-d) ≈⟨ ⟩≈ bitonic-sorter-4)
  let (bit4-out-1 , bit4-next-1) = runPG bitonic-sorter-4 (test-pulse-a)
      (bit4-out-2 , bit4-next-2) = runPG bit4-next-1 (test-pulse-b)
      (bit4-out-3 , bit4-next-3) = runPG bit4-next-2 (test-pulse-c)
      (bit4-out-4 , bit4-next-4) = runPG bit4-next-3 (test-pulse-d)
  in (proj₁ bit4-out-1) ≡ (* :: ∅ :: ∅ :: ∅ :: []) ×
     (proj₁ bit4-out-2) ≡ (∅ :: * :: ∅ :: ∅ :: []) ×
     (proj₁ bit4-out-3) ≡ (∅ :: ∅ :: * :: ∅ :: []) ×
     (proj₁ bit4-out-4) ≡ (∅ :: ∅ :: ∅ :: * :: []) ×
     bit4-next-4 ≈⟨ ⟩≈ bitonic-sorter-4
proj₁ (bitonic-correct coind) = refl
proj₁ (proj₂ (bitonic-correct coind)) = refl
proj₁ (proj₂ (proj₂ (bitonic-correct coind))) = refl
proj₁ (proj₂ (proj₂ (proj₂ (bitonic-correct coind)))) = refl
proj₂ (proj₂ (proj₂ (proj₂ (bitonic-correct coind)))) = coind
```

Listing 24. The 4-wide bitonic sorter correctness proof.

by Citrus, the proofs are straightforward. We further note that the PyLSE paper attempted the 8-bit bitonic sorter as well but similarly it succumbed to state explosion and was not able to be model checked. We can see from the above code that extension to the 8-bit case is trivial and correctness can be proven in an identical manner.

5.5 Discussion

Citrus, establishes an idealized model in which it proves certain fundamental laws about SCE. They are interesting for building up our knowledge about SCE as a unique information encoding and constitute basic optimizations in SCE workflows which could be taken advantage of by designers using SPICE, Verilog, etc. The field of SCE is still scaling and so the breadth of moderately large circuits with documented failures is limited. That is, by traditional CMOS standards, a 4-wide bitonic sorter would be miniscule, however, in the context of SCE, this is still a relatively interesting circuit. Indeed, the types of circuits that SCE should be targeting is itself an open question. For instance, the work of Tzimpragos on race trees in [Tzimpragos et al. 2019] is an exploration of what kinds of circuits SCE are well-designed to represent.

This brings up an interesting issue concerning what can be verified in Citrus versus PyLSE, e.g. a model checker. In terms of safety and liveness, model checking, specifically the language of Timed Computation Tree Logic used in Uppaal, should be able to at least formulate any safety property (though scalability issues can prevent checking them). The correctness properties given in PyLSE are all safety properties. The move to coinductive types does open the door to checking liveness

properties as well. This is an avenue we are interested in exploring, formulating a wider spectrum of properties and addressing these as well.

6 Related Work

The most closely related work is PyLSE [Christensen et al. 2022], which we have described previously in the paper. Here we discuss other related works besides PyLSE.

6.1 High-Level Hardware Design

Some of the earliest approaches to combining functional programming and hardware description come in the form of Lava [Bjesse et al. 1998] and Hawk [Matthews et al. 1998]. Lava and Hawk are Haskell-based DSLs for describing and modeling synchronous digital hardware. They offer certain amounts of formal verification for the behavior of circuits in addition to creating an environment for more compositional hardware design. However, both Lava and Hawk are designed for conventional CMOS circuits rather than the asynchronous pulse-based circuits required for SCE and hence are not usable for describing SCE circuits.

Hawk and Lava both support certain kinds of verification; however, neither treat circuit equivalence the way Citrus does and neither uses the notion of arrows. Hawk supports model checking via binary decision diagrams [Launchbury et al. 1999]. Lava supports model checking against computational tree logic (CTL) formulas, as well as facilities for SMT solving and for proving safety formulas using an inductive proof strategy [Claessen and Sheeran 2000]. As the approach is restricted to safety properties, it is only valid for properties defined as least fixed points and not for properties defined as greatest fixed points, e.g., coinductively, in contrast to Citrus. The Kansas dialect of Lava is a more recent extension of the original Lava code and enables writing equalities at the type level, but only between types representing finite sets [Gill et al. 2010].

6.2 Functional Reactive Programming and Hardware Design

Functional reactive programming (FRP) is an approach to designing persistent, stateful systems based on a small set of function combinators [Elliott and Hudak 1997a]. In modern approaches these combinators are built on top of arrows [Paterson 2003]. There have been a handful of attempts to describe basic hardware circuitry as reactive systems, specifically within the framework of arrow-based FRP. The Cλash [Baaij 2009] system provides a DSL implemented in the Haskell language for describing synchronous circuits. Cλash is more of a designer's tool and does not focus on formal verification, but it does have an experimental API that allows for expressing logical formulas that are translated into Property Specification Language (PSL), a variant of LTL, and SystemVerilog Assertions. Π-ware [Flor et al. 2018] is a DSL written in Agda that expresses traditional CMOS synchronous circuits and offers methods for testing equivalence as well as synthesis. It does not offer facilities for timing and its notion of equivalence is based on finite simulation, e.g. checking finite, equal outputs for finite, equal inputs, which is not useful for SCE circuit equivalence.

Outside of hardware design, [Yallop and Liu 2016] explore an Arrow type which further restricts the ArrowLoop typeclass to contain only those arrows which are causal and commutative. These are two restrictions on the sequential and parallel composition operators, as well as a restriction on the loop operator. The result of this restricted typeclass is that the authors can use a series of equations to reduce any given causal, commutative arrow (CCA) into a single arrow of a certain form. The CCA arrows bear some resemblance to our timed arrows. The CCA typeclass include a delay combinator, but this is less about timing and more about restricting Haskell's $\perp$ type during feedback computations. Nevertheless, we speculate that we can employ a similar normalization algorithm and thus obtain an automatic form of equivalence checking, namely, computing two

arrows to their normal form and checking if they are exactly equal. We leave this activity to future work.

Finally, the work of Bärenz is also outside the realm of hardware but considers adding type-level clocks and coinductive tooling to FRP combinators in [Bärenz and Perez 2018; Bärenz and Seufe 2017]. While Citrus has not focused specifically on synchronous, clock-based circuits, the developments in these two works could provide an effective way toward this goal.

7 Conclusion

We have presented Citrus, an embedded DSL in Agda for modeling SCE circuits as functional objects, and described a useful notion of equivalence for Citrus programs that enables reasoning about the equivalence of SCE circuits up to time-abstraction. Citrus offers a platform for exploring SCE circuit transformations which can lead to provably-correct circuit optimizations. Moreover, the core language is easily extendable with new combinators and effects. We evaluated Citrus on two sets of case studies. In the first we proved several equational laws over SCE circuits which have never been formalized. In the second, we demonstrated a proof of correctness for a moderately sized SCE circuit which was unable to be model checked in PyLSE. We believe Citrus greatly expands on the programming language toolbox for SCE that is available to hardware architects and designers.

References

Andreas Abel. 2010. MiniAgda: Integrating Sized and Dependent Types. Electronic Proceedings in Theoretical Computer Science 43 (dec 2010), 14–28. doi:10.4204/eptcs.43.2

Andreas Abel. 2016. Equational Reasoning about Formal Languages in Coalgebraic Style. (2016), 38. https://www.cse.chalmers.se/~abela/jlamp17.pdf

Luca Aceto, Anna Ingolfsdottir, Kim Guldstrand Larsen, and Jiri Srba. 2007. Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, Cambridge. doi:10.1017/CBO9780511814105

Rajeev Alur and David L. Dill. 1994. A theory of timed automata. Theoretical Computer Science 126, 2 (apr 1994), 183–235. doi:10.1016/0304-3975(94)90010-8

C. Baaij. 2009. CλasH : from Haskell to hardware. http://essay.utwente.nl/59482/

Manuel Bärenz. 2020. The essence of live coding: change the program, keep the state!. In Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (Virtual, USA) (REBLS 2020). Association for Computing Machinery, New York, NY, USA, 2–14. doi:10.1145/3427763.3428312

Manuel Bärenz and Ivan Perez. 2018. Rhine: FRP with type-level clocks. SIGPLAN Not. 53, 7 (Sept. 2018), 145–157. doi:10.1145/3299711.3242757

Manuel Bärenz and Sebastian Seufe. 2017. Verifying Functional Reactive Programs with Side Effects. (2017). https://types2017.elte.hu/proc.pdf#page=47 TYPES 2017.

Johan Bengtsson and Wang Yi. 2004. Timed Automata: Semantics, Algorithms and Tools. In Lectures on Concurrency and Petri Nets, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg (Eds.). Vol. 3098. Springer Berlin Heidelberg, Berlin, Heidelberg, 87–124. doi:10.1007/978-3-540-27755-2_3 Series Title: Lecture Notes in Computer Science.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. SIGPLAN Not. 34, 1 (sep 1998), 174–184. doi:10.1145/291251.289440

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda – A Functional Language with Dependent Types. In Theorem Proving in Higher Order Logics, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–78. doi:10.1007/978-3-642-03359-9_6

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 (San Diego, CA, USA) (PLDI 2022). Association for Computing Machinery, New York, NY, USA, 671–686. doi:10.1145/3519939.3523438

Koen Claessen and Mary Sheeran. 2000. A Tutorial on Lava: A Hardware Description and Verification System. (10 2000).

Johannes Arnoldus Delport, Kyle Jackman, Paul le Roux, and Coenrad Johann Fourie. 2019. JoSIM—Superconductor SPICE Simulator. IEEE Transactions on Applied Superconductivity 29, 5 (2019), 1–5. doi:10.1109/TASC.2019.2897312

Conal Elliott and Paul Hudak. 1997a. Functional Reactive Animation. SIGPLAN Not. 32, 8 (aug 1997), 263–273. doi:10.1145/258949.258973

Conal Elliott and Paul Hudak. 1997b. Functional Reactive Animation. In Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (Amsterdam, The Netherlands) (ICFP '97). Association for Computing Machinery, New York, NY, USA, 263–273. doi:10.1145/258948.258973

João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. 2018. Pi-Ware: Hardware Description and Verification in Agda. In 21st International Conference on Types for Proofs and Programs (TYPES 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 69), Tarmo Uustalu (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 9:1–9:27. doi:10.4230/LIPIcs.TYPES.2015.9

Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. 2010. Introducing Kansas Lava. In Implementation and Application of Functional Languages, Marco T. Morazán and Sven-Bodo Scholz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–35.

Masahito Hasegawa. 1999. Recursion from Cyclic Sharing. Springer London, London, 83–101. doi:10.1007/978-1-4471-0865-8_7

H. Hayakawa, N. Yoshikawa, S. Yorozu, and A. Fujimaki. 2004. Superconducting digital electronics. Proc. IEEE 92, 10 (oct 2004), 1549–1563. doi:10.1109/JPROC.2004.833658

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

Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. Arrows, Robots, and Functional Reactive Programming. Springer Berlin Heidelberg, Berlin, Heidelberg, 159–187. doi:10.1007/978-3-540-44833-4_6

John Hughes. 2000. Generalising monads to arrows. Science of Computer Programming 37, 1 (2000), 67–111. doi:10.1016/S0167-6423(99)00023-4

John Hughes. 2005. Programming with Arrows. In Advanced Functional Programming, Varmo Vene and Tarmo Uustalu (Eds.). Vol. 3622. Springer Berlin Heidelberg, Berlin, Heidelberg, 73–129. doi:10.1007/11546382_2 Series Title: Lecture Notes in Computer Science.

Bart Jacobs. 2016. Introduction to Coalgebra: Towards Mathematics of States and Observation. Cambridge University Press.

John Launchbury, Jeffrey R. Lewis, and Byron Cook. 1999. On Embedding a Microarchitectural Design Language within Haskell. In Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (Paris, France) (ICFP '99). Association for Computing Machinery, New York, NY, USA, 60–69. doi:10.1145/317636.317784

Hai Liu, Eric Cheng, and Paul Hudak. 2009. Causal Commutative Arrows and Their Optimization. SIGPLAN Not. 44, 9 (aug 2009), 35–46. doi:10.1145/1631687.1596559

John Matthews, Byron Cook, and John Launchbury. 1998. Microprocessor Specification in Hawk. 90–101. doi:10.1109/ICCL.1998.674160

Ross Paterson. 2001. A New Notation for Arrows. In International Conference on Functional Programming (Firenze, Italy). ACM Press, 229–240. https://www.staff.city.ac.uk/~ross/papers/notation.pdf

Ross Paterson. 2003. Arrows and computation. In The Fun of Programming, Jeremy Gibbons and Oege de Moor (Eds.). Macmillan Education UK, London, 201–222. doi:10.1007/978-1-349-91518-7_10

Christine Paulin-Mohring. 2012. Introduction to the Coq Proof-Assistant for Practical Software Verification. Springer Berlin Heidelberg, Berlin, Heidelberg, 45–95. doi:10.1007/978-3-642-35746-6_3

Jan Rutten. 2019. The Method of Coalgebra: exercises in coinduction. https://ir.cwi.nl/pub/28550/

Davide Sangiorgi. 2011. Introduction to Bisimulation and Coinduction. Cambridge University Press, Cambridge. doi:10.1017/CBO9780511777110

I. I. Soloviev, N. V. Klenov, S. V. Bakurskiy, M. Yu. Kupriyanov, A. L. Gudkov, and A. S. Sidorenko. 2017. Beyond Moore's technologies: operation principles of a superconductor alternative. arXiv e-prints, Article arXiv:1706.09124 (June 2017), arXiv:1706.09124 pages. arXiv:1706.09124 [cond-mat.supr-con] doi:10.48550/arXiv.1706.09124

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. doi:10.1145/3297858.3304036

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. doi:10.1109/MM.2021.3066377

Zhanyong Wan and Paul Hudak. 2000. Functional reactive programming from first principles. SIGPLAN Not. 35, 5 (May 2000), 242–252. doi:10.1145/358438.349331

Jeremy Yallop and Hai Liu. 2016. Causal Commutative Arrows Revisited. In Proceedings of the 9th International Symposium on Haskell (Nara, Japan) (Haskell 2016). Association for Computing Machinery, New York, NY, USA, 21–32. doi:10.1145/

2976002.2976019

Received 2026-02-18; accepted 2026-05-13