Implicit State Machines

Anonymous Authors

Abstract

Finite-state machines (FSM) are a simple yet powerful abstraction widely used for modeling, programming and verifying real-time and reactive systems that control modern factories, power plants, transportation systems and medical equipment.

However, traditionally finite-state machines are either encoded indirectly in an *imperative* language, such as C and Verilog, or embedded as an *imperative* extension of a declarative language, such as Lustre. Given the widely accepted advantage of declarative programming, can we have a declarative design of finite-state machines to facilitate design, construction, and verification of embedded programs?

By sticking to the design principle of declarativeness, we show that a novel abstraction emerges, *implicit state machines*, which is declarative in nature and at the same time supports recursive composition. Given its simplicity and universality, we believe it may serve as a new foundation for programming embedded systems.

CCS Concepts: • **Hardware → Software tools for EDA**; • **Software and its engineering → Domain specific languages**.

Keywords: Finite-state machines, hierarchical finite-state machines, implicit state machines

ACM Reference Format:

Anonymous Authors. 2022. Implicit State Machines. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Finite-state machines are a universal formalism for modeling, programming and verifying real-time and reactive systems, and are widely used in industiral automation, public transportation systems, medical equipment, as well as avionics.

While many imperative languages, such as C and Verilog, support finite-state machines via encoding, it is more advantageous to have finite-state machines as a language construct to facilitate design, construction, and verification of embedded systems.

However, most such programming models of finite-state machines are in an *imperative* style, instead of being *declarative* [Colaço et al. 2005; Li et al. 2011]. It is well known that declarative programming bridges the gap of specification and implementation, facilitates program transformation and verification, and at the same time less error-prone than imperative programming [Fernández et al. 2000; Halbwachs et al. 1989; Hanus and Kluß 2009; Hinrichs 2011; Leuschel 2008; Søndergaard and Sestoft 1990]. We therefore ask the following question:

> **Can we have a declarative design of finite-state machines to facilitate design, construction, and verification of embedded systems?**

The main idea of this paper is to show that by adhering to the design principle of declarativeness, we discover a novel abstraction, which we call *implicit state machines*, and which answers the question above affirmatively.

Our contributions are listed below:

- Following the design principle of declarativeness, we discover a novel abstraction, *implicit state machines*, which are simple, flexible, universal and recursively composable.
- We formalize the concept of implicit state machines in a core calculus specialized with the domain of Boolean algebra. We show that it can serve as a simple and elegant model for sequential digital circuits, which is not known previously.
- We develop an embedded DSL in Scala for digital design based on implicit state machines, and we demonstrate its practicality by designing a micro-controller in the DSL.

2 Deriving Implicit State Machines

Readers may skip this section and jump to the intuitive introduction of implicit state machines on the first read (Section 3).

Mathematically, a finite state machine is usually represented as a quintuple (I, S, s_0, σ, O) [1]:

- I is the set of inputs;
- S is the set of states;
- $s_0 \in S$ is the initial state;
- $\sigma : I \times S \to S \times O$ maps the input and the current state to the next state and the output;
- O is the set of outputs.

[1] Technically, the quintuple described here is a Mealy machine, because it has an output. In embedded systems, pristine FSMs without output are not interesting.

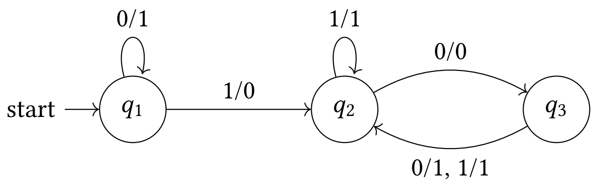
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, July 2017, Washington, DC, USA

© 2022 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

FSM can also be represented graphically by *state-transition diagrams*, as the following figure shows:



In the state machine above, q_1 is the initial state, and each edge denotes a state transition: the label 0/1 on the edge means the transition happens when the input is 0, and it outputs 1 when the transition occurs.

Implicit state machines are based on a reflection on the essence of FSMs: a mapping from input and state to the next state and output.

Insight 1. The first insight towards implicit state machines is that *state transitions do not need to be explicitly enumerated*, as it is taken for granted in existing languages for programming with FSMs [Caspi and Pouzet 2008; Colaço et al. 2005; Harel 1987; Li et al. 2011].

In a declarative language, the mapping can be represented by any expression. This gives us a tentative representation as follows:

$$\lambda x{:}I \times S.\, (t_1, t_2) \qquad : \qquad I \times S \rightarrow S \times O$$

The body (t_1, t_2) enforces that the output and next state are implemented as two functions. This imposes unnecessary syntatic constraints. If we introduce tuples in the language, we can replace (t_1, t_2) just by t:

$$\lambda x{:}I \times S.\, t \qquad : \qquad I \times S \rightarrow S \times O$$

Insight 2. The second insight is that *the state is neither an input to an FSM nor an output of an FSM, but an internal value*. It leads us to the following representation with the state variable s:

$$\lambda x{:}I.\, \mathit{fsm}\ \{\ s \Rightarrow t\ \} \qquad : \qquad I \rightarrow O$$

In the above, the term t still has the type $S \times O$. But seen from outside, a state machine just maps input to output, which corresponds to our intuition.

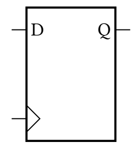
Insight 3. The last insight is that *the inputs do not need to be declared explicitly*, they can be *captured* from the lexical scope, similar to capture in lambda calculus [Barendregt 1985]:

$$\mathit{fsm}\ \{\ s \Rightarrow t\ \} \qquad : \qquad O$$

We still miss the initial state, so we use the value v to denote the initial state of the FSM:

$$\mathit{fsm}\ \{\ v\ \mid\ s \Rightarrow t\ \} \qquad : \qquad O$$

Voila! We arrived at a declarative representation of finite-state machines.



(a) Circuit Symbol

S	D	S'	Q
0	0	0	0
0	1	1	0
1	0	0	1
1	1	1	1

(b) Truth table

Figure 1. D flip-flops and its semantics

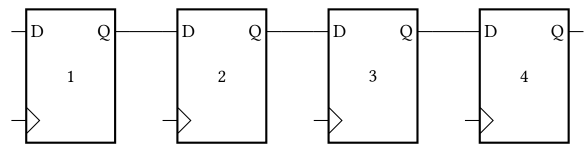


Figure 2. A 4-bit serial-in serial-out shift register

3 Implicit State Machines, Informally

Suppose we are working in the domain of digital circuits. One of the most common state elements in digital circuits are *D flip-flops*, whose symbol and truth-table semantics are presented in Figure 1.

Intuitively, D flip-flops delay the input D by one clock. It can be seen from the truth table that the next state S' is always equal to the input D, and the output Q is always equal to the current state S.

Using implicit state machines, a one-bit D flip-flop with an input signal d can be represented as follows:

$$\mathit{fsm}\ \{\ 0\ \mid\ s \Rightarrow (d, s)\ \}$$

In the above, 0 represents the initial state of the D flip-flop; s represents the current state; d represents the input. The body is a pair (d, s), which means that the next state of the implicit state machine is the input d and the output is the current state s.

Note that in the above, the state variable s is bound, while the input d is not bound. This is a characteristic of implicit state machines, where the inputs are implicit, i.e., they are captured from the lexical environment, similar to capture in lambda calculus [Barendregt 1985].

D flip-flops can be use to implement shift registers. In Figure 2, we implement a 4-bit serial-in serial-out shift register by chaining 4 D flip-flops. As a single D flip-flop delays the input signal by one clock, intuitively the 4-bit serial-in serial-out shift register delays the input signal by 4 clocks.

We implement the 4-bit serial-in serial-out shift register for a given input d with implicit state machine as follows:

```
let q1 = fsm { 0 | s => (d, s)  } in
let q2 = fsm { 0 | s => (q1, s) } in
let q3 = fsm { 0 | s => (q2, s) } in
let q4 = fsm { 0 | s => (q3, s) } in
q4
```

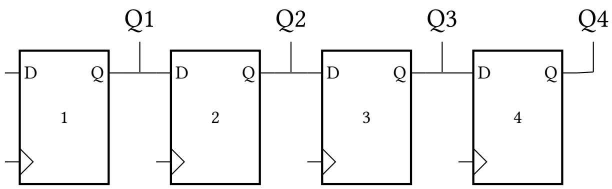


Figure 3. A 4-bit serial-in parallel-out shift register

In the code above, we use the standard linguistic construct *let*/*in* to introduce local bindings.

Implicit state machines are just expressions, thus they may appear in any place where an expression is allowed. In particular, we may nest them to get another equivalent implementation of the 4-bit serial-in serial-out shift register:

```
fsm { 0 | s =>
  let q1 = fsm { 0 | s => (d, s) } in
  let q2 = fsm { 0 | s => (q1, s) } in
  let q3 = fsm { 0 | s => (q2, s) } in
  (q3, s)
}
```

An equivalent and simpler implementation of the 4-bit serial-in serial-out shift register is shown below:

```
fsm { (0, 0, 0, 0) | s => ((d, s.1, s.2, s.3), s.4) }
```

In the above, we use the syntax $(t, \ldots, t)$ to represent a tuple, and $t.i$ to represent the i-th component of the tuple t. In fact, we will show in the next section, there is a mechanic transformation from all other equivalent representation to this succinct form (Section 4.4).

There are also serial-in parallel-out shift registers, as shown in Figure 3. They can be implemented with implicit state machines as follows:

```
let q1 = fsm { 0 | s => (d, s)  } in
let q2 = fsm { 0 | s => (q1, s) } in
let q3 = fsm { 0 | s => (q2, s) } in
let q4 = fsm { 0 | s => (q3, s) } in
(q1, q2, q3, q4)
```

An equivalent and simpler implementation of the 4-bit serial-in parallel-out shift register is shown below:

```
fsm { (0, 0, 0, 0) | s => ((d, s.1, s.2, s.3), s) }
```

We draw the readers' attention to the following properties of implicit state machines.

Declarativeness. In contrast to existing *imperative* programming models with finite-state machines [Colaço et al. 2005; Li et al. 2011], implicit state machines are *declarative*. As we will see in the next section, the declarative nature of implicit state machines facilitate transformation of programs, thanks to *referential transparency*, which enables *substitute equals for equals* [Søndergaard and Sestoft 1990].

Simplicity. As we have seen in the example of D flip-flops, compared to the textbook presentation of D flip-flops in the form of truth tables, the representation based on implicit state machines is much simpler and more intuitive. While simple concepts can be explained in a complex way, we do not see how implicit state machines could be reduced to a simpler model.

Flexibility. Most existing programming models with finite state machines demand explicit enumeration of state transitions [Caspi and Pouzet 2008; Colaço et al. 2005; Harel 1987; Li et al. 2011]. In contrast, implicit state machines just do not mandate explicit enumeration of state transitions in the program. However, they do not forbid that. This means that programmers can continue to program with explicit states when necessary. This is can be done by introducing a *match*-expression:

```
fsm { 0 | s =>
  match (s, d) with
  case (0, 0) => (0, 0)
  case (0, 1) => (1, 0)
  case (1, 0) => (0, 1)
  case (1, 1) => (1, 1)
}
```

In the above, the `match` expression defines the semantics of D flip-flops in the form of truth tables (Figure 1).

Recursive Composability. Implicit state machines are recursively composable, which is a yardstick of proper language design. Recursive composability corresponds to the need for hierarchical decomposition in designing real-world systems.

Universality. The universality of implicit state machines are inherited from the universality of finite-state machines, as the latter can be represented by the former.

4 Implicit State Machines, Formally

In this section, we formalize implicit state machines in a small calculus with Boolean algebra as the domain intended for digital design.

4.1 Syntax

The syntax of the calculus is presented below:

t	$::=$		*terms*
		a, b, c	*external input*
		x, y, z, s	*variables*
		$let\ x = t\ in\ t$	*let binding*
		β	*Boolean value*
		$t * t$	*1 bit and*
		$t + t$	*1 bit or*
		$!t$	*1 bit not*
		$(t, \ldots, t)$	*tuple*
		$t.i$	*projection*
		$fsm\ \{\ v\ \mid\ s \Rightarrow t\ \}$	*implicit state machine*
β	$::=$	$0 \mid 1$	*Boolean values*
v	$::=$	$\beta \mid (v, \ldots, v)$	*values*
i	$::=$	$0, 1, 2, \ldots$	*indexes*

Beyond the basic elements of Boolean algebra, we also introduce *let*-bindings, which is a basic abstraction and reuse mechanism. Tuples and projections are introduced for parallel composition and decomposition. In a projection $t.i$, the index i must be a statically known number. For implicit state machines, we require that the initial state is a value.

A circuit usually has external inputs, which are represented by variables a, b, c. By convention, we use x, y, z for *let*-bindings, and s for the binding in implicit state machines.

We choose Boolean algebra as the domain theory, but it can also be other mathematical structures, for example natural numbers or tensors. Our transform does not assume properties of mathematical structures as long as we may *substitute equals for equals* [Søndergaard and Sestoft 1990].

To avoid technical details of same names in bindings, we assume the uniqueness of bound variables, which can be easily achieved via renaming.

4.2 Semantics

The semantics of the language is defined with the help of a state σ and an environment ρ. The state σ maps a state variable to a state value, the environment variable ρ maps an external input to a value. The big-step operational semantics is defined with the following reduction relation:

$$t \xrightarrow{\sigma,\rho} v \mid \sigma'$$

It means that given the current state σ and environment ρ, the term t evaluates to the value v with the next state σ'. The semantics follows the *synchronous hypothesis* [Benveniste and Berry 1991], which assumes that the computation of the response to an input takes no time. For synchronous digital circuits, it means that the system produces an output at each clock tick. The reduction rules are defined in Figure 4. We explain the rules below:

- E-Value. If the term is already a value, do nothing. There are no nested state machines, thus the mapping for the next state is the empty set.
- E-Input. Look up the external variable a from the environment ρ.
- E-Let. First evaluate t_1 to the value v_1, then evaluate t_2 with x replaced by v_1.
- E-Tuple. Evaluate each component in parallel to a value, and accumulate the mapping for the next state.
- E-Project. First evaluate the term to a tuple value, then return the corresponding component.
- E-And. Evaluate the two components in parallel to Boolean values, then call the helper method *and* to compute the resulting Boolean value β. As each component may contain implicit state machines, accumulate the mapping for the next state.
- E-Or. Similar as above, but use the helper function *or* to compute the resulting value.
- E-Not. Similar as above, but use the helper function *not* to compute the resulting value.
- E-Fsm. First look up the value for the current state from the state map σ. Then evaluate the body of the state machine to a pair value (v_1, v_2). The output is v_2, and the next state of the FSM is v_1.

The reduction relation only defines one-tick semantics. The semantics of a system is defined by the *trace* of a given input series $\rho_0, \rho_1, \cdots$. We define it formally below:

Definition 4.1 (Trace)**.** The trace of a system t with respect to an input sequence $\rho_0, \rho_1, \cdots$ is the sequence $o_0, o_1, \cdots$ such that

- $t \xrightarrow{\sigma_0,\rho_0} o_0 \mid \sigma_1$
- $\cdots$
- $t \xrightarrow{\sigma_i,\rho_i} o_i \mid \sigma_{i+1}$
- $\ldots$

In the above, σ_0 is the initial state of the implicit state machines as specified in t.

4.3 Type System

To check well-formedness of programs, we introduce a simple type system to ensure that a well-typed program never gets stuck. The type system is presented in Figure 5.

In the system, there are two types: *Bool* for Boolean values and $(T_1, \ldots, T_n)$ for tuples. We explain the typing rules below:

- T-Bool. The type for Boolean values is always *Bool*.
- T-Input. For inputs, their types are predefined in the environment.
- T-Var. For variables, their types also appear in the environment.
- T-Not. The term t must be of the type *Bool*.
- T-Tuple. If each component has a type, and then the type of the tuple has a corresponding tuple type.
- T-Project. If the term t has a tuple type, then the projection has the type of the corresponding component.
- T-And. If each component has the type *Bool*, the result also has the type *Bool*.
- T-Or. The same as above.

$$v \xrightarrow{\sigma,\rho} v \mid \emptyset \quad \text{(E-VALUE)} \qquad\qquad a \xrightarrow{\sigma,\rho} \rho(a) \mid \emptyset \quad \text{(E-INPUT)}$$

$$\frac{t_1 \xrightarrow{\sigma,\rho} v_1 \mid \sigma' \qquad [x \mapsto v_1]t_2 \xrightarrow{\sigma,\rho} v_2 \mid \sigma''}{let\ x = t_1\ in\ t_2 \xrightarrow{\sigma,\rho} v \mid \sigma' \cup \sigma''} \quad \text{(E-LET)}$$

$$\frac{t_1 \xrightarrow{\sigma,\rho} v_1 \mid \sigma_1 \qquad \dots \qquad t_n \xrightarrow{\sigma,\rho} v_n \mid \sigma_n}{(t_1, \dots, t_n) \xrightarrow{\sigma,\rho} (v_1, \dots, v_n) \mid \sigma_1 \cup \dots \cup \sigma_n} \quad \text{(E-TUPLE)}$$

$$\frac{t \xrightarrow{\sigma,\rho} (v_1, \dots, v_i, \dots, v_n) \mid \sigma'}{t.i \xrightarrow{\sigma,\rho} v_i \mid \sigma'} \quad \text{(E-PROJECT)}$$

$$\frac{t_1 \xrightarrow{\sigma,\rho} \beta_1 \mid \sigma' \qquad t_2 \xrightarrow{\sigma,\rho} \beta_2 \mid \sigma'' \qquad \beta = and(\beta_1, \beta_2)}{t_1 * t_2 \xrightarrow{\sigma,\rho} \beta \mid \sigma' \cup \sigma''} \quad \text{(E-AND)}$$

$$\frac{t_1 \xrightarrow{\sigma,\rho} \beta_1 \mid \sigma' \qquad t_2 \xrightarrow{\sigma,\rho} \beta_2 \mid \sigma'' \qquad \beta = or(\beta_1, \beta_2)}{t_1 + t_2 \xrightarrow{\sigma,\rho} \beta \mid \sigma' \cup \sigma''} \quad \text{(E-OR)}$$

$$\frac{t \xrightarrow{\sigma,\rho} \beta \mid \sigma' \qquad \beta' = not(\beta)}{!t \xrightarrow{\sigma,\rho} \beta' \mid \sigma'} \quad \text{(E-NOT)}$$

$$\frac{v = \sigma(s) \qquad [s \mapsto v]t \mid \xrightarrow{\sigma,\rho} (v_1, v_2) \mid \sigma'}{fsm\ \{\ v_0 \ \mid\ s \Rightarrow t\ \} \xrightarrow{\sigma,\rho} v_2 \mid \{\ s \mapsto v_1\ \} \cup \sigma'} \quad \text{(E-FSM)}$$

Figure 4. Big-step operational semantics

$$T ::= Bool \mid (T, \dots, T)$$

$$\Gamma \vdash \beta : Bool \quad \text{(T-BOOL)} \qquad\qquad \frac{\Gamma \vdash t : (T_1, \dots, T_i, \dots, T_n)}{\Gamma \vdash t.i : T_i} \quad \text{(T-PROJECT)}$$

$$\frac{a : T \in \Gamma}{\Gamma \vdash a : T} \quad \text{(T-INPUT)} \qquad\qquad \frac{\Gamma \vdash t_1 : Bool \qquad \Gamma \vdash t_2 : Bool}{\Gamma \vdash t_1 * t_2 : Bool} \quad \text{(T-AND)}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)} \qquad\qquad \frac{\Gamma \vdash t_1 : Bool \qquad \Gamma \vdash t_2 : Bool}{\Gamma \vdash t_1 + t_2 : Bool} \quad \text{(T-OR)}$$

$$\frac{\Gamma \vdash t : Bool}{\Gamma \vdash !t : Bool} \quad \text{(T-NOT)} \qquad\qquad \frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash let\ x = t_1\ in\ t_2 : T_2} \quad \text{(T-LET)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \dots \qquad \Gamma \vdash t_n : T_n}{\Gamma \vdash (t_1, \dots, t_n) : (T_1, \dots, T_n)} \quad \text{(T-TUPLE)} \qquad\qquad \frac{\Gamma \vdash v : T_1 \qquad \Gamma, s{:}T_1 \vdash t : (T_1, T_2)}{\Gamma \vdash fsm\ \{\ v \ \mid\ s \Rightarrow t\ \} : T_2} \quad \text{(T-FSM)}$$

Figure 5. Type System

- T-LET. If the bound term has the type of T_1, and the body of the let-binding has the type T_2 under the environment Γ extended with the binding $x{:}T_1$, then the let-binding has the type T_2. Note that this rule forbids the usage of x in t_1, which prevents undesired circles.

- T-FSM. If the initial value has the type T_1, and the body has the type (T_1, T_2) under the environment Γ extended with the binding $s{:}T_1$, then the implicit state machine has the type T_2.

For the meta-theory of the type system, we need to define well-formedness of the input map and state map. We write $\Gamma \vdash \xi$ to mean that the input map or state map ξ is well-typed under Γ, which is defined as follows:

$$\Gamma \vdash \emptyset \qquad\qquad \frac{\Gamma \vdash \xi \qquad \emptyset \vdash v : T}{\Gamma, \alpha{:}\, T \vdash \xi \cup \{\, \alpha \mapsto v \,\}}$$

In the above, α ranges over input and state variables, and ξ ranges over input map and state map.

Theorem 4.2 (Soundness)**.** *If $\Gamma \vdash t : T$, and if for each ρ_i in the input sequence $\rho_0, \rho_1, \ldots$ we have $\Gamma \vdash \rho_i$, then there exists a trace for the system t corresponding to the input sequence.*

The proof follows from the following lemma by induction on the length of the input sequence:

Lemma 4.3. *If t is well-typed under the environment Γ, and the input map ρ is compatible with Γ, and the state map σ is type-compatible with the initial state map σ_0 as specified in t, then t evaluates to a value v with updated state map σ'.*

More formally, if $\Gamma \vdash t : T$, and $\Gamma \vdash \rho$, and there exists Γ' such that $\Gamma' \vdash \sigma$ and $\Gamma' \vdash \sigma_0$, then there exists v and σ' such that $t \xrightarrow{\sigma,\rho} v \mid \sigma'$, $\Gamma \vdash v : T$ and $\Gamma' \vdash \sigma'$.

Sketch. By induction on the typing judgment $\Gamma \vdash t : T$. □

4.4 Flattening

In this section, we show that any system of implicit state machines is equivalent to a single flat implicit state machine. This can be achieved by a mechanic transformation.

For the purpose of the transformation, we first define the *combinational fragment* of the language devoid of implicit state machines, which is represented by e:

$$\begin{aligned} e \quad ::= \quad & \beta \mid e * e \mid e + e \mid !e \mid (e, \ldots, e) \mid e.i \mid \\ & let\ x = e\ in\ e \mid x \mid s \mid a \end{aligned}$$

The combinational fragment corresponds to combinational circuits, i.e., circuits without state elements, in contrast to sequential circuits.

The transformation consists of two major steps:

- **Lifting**: lifts FSMs to top-level (Figure 6).
- **Merging**: merges FSMs to a single FSM (Figure 7).

Lifting (Figure 6) results in *lifted normal form* (u) where all FSMs are nested at the top-level of the program, with a combinational fragment in the middle:

$$u \quad ::= \quad e \mid fsm\,\{\, v \mid s \Rightarrow u \,\}$$

The relation $t_1 \rightsquigarrow_L t_2$ says that the term t_1 takes a lifting step to t_2. Lifting is defined with the help of the lifting context L. The lifting context specifies that the transformation follows the order left-right and top-down. The actual lifting happens with the function $[\![\cdot]\!]$, which transforms the source program to the expected form. We explain the concrete transformation rules below:

- $fsm\ \{\ v \mid s \Rightarrow e_1\ \} * t_2$. The FSM absorbs t_2 into its body. The symmetric case, and the cases for AND and OR are similar.

- $let\ x = fsm\ \{\ v \mid s \Rightarrow e_1\ \}\ in\ t_2$. It pulls the let-binding into the body. The case in which FSM is in the body of let-binding is similar.

- $fsm\ \{\ v \mid s \Rightarrow e\ \}.i$. It pulls the projection into the body of FSM.

- $(\bar{e}, fsm\ \{\ v \mid s \Rightarrow e\ \}, \bar{t})$. It pulls the tuple into the body of FSM.

Note that to simplify the presentation, in the transformation rules we write $let\ x, y = t_1\ in\ t_2$ as a syntactic sugar for $let\ z = t_1\ in\ let\ x = z.1\ in\ let\ y = z.2\ in\ t_2$.

Once all FSMs are nested at the top-level after lifting, merging (Figure 7) takes place. The relation $u_1 \rightsquigarrow_M u_2$ says that the term u_1 takes a merging step to u_2. Merging is defined with the help of the merging context M. The merging context specifies that the merging happens from inside towards outside. The actual merging step is quite straightforward: it just combines the initial states v_1 and v_2, as well as merges s_1 and s_2 into s.

4.5 Discussion

Flattening makes it immediately obvious that a digital circuit with state elements (such as registers and flip-flops) are equivalent to a combinational circuit with all state elements at the boundary. This enables *global* optimization of circuits using combinational techniques.

We believe the insight itself is not new, however, implicit state machines make it obvious. In contrast, it is obscured in the network-based model of digital circuits, e.g., it is not obvious how to push a D flip-flop in the middle of a circuit network to its boundary.

The declarative nature of implicit state machines enables the reasoning principle of *substituting equals for equals* [Søndergaard and Sestoft 1990]. It facilitates many common program optimizations, such as dead-code elimination, common-subexpression elimination, constant folding, etc.

$$L \quad ::= \quad [\cdot] \;\mid\; L * t \;\mid\; e * L \;\mid\; L + t \;\mid\; e + L \;\mid\; !L \;\mid\; L.i \;\mid\; (e_1, \ldots, L, \ldots, t_n) \;\mid$$
$$\mathit{fsm}\ \{\ v \;\mid\; s \Rightarrow L\ \} \;\mid\; \mathit{let}\ x = L\ \mathit{in}\ t \;\mid\; \mathit{let}\ x = e\ \mathit{in}\ L$$

$$\frac{[\![t]\!] = \mathit{fsm}\ \{\ v \;\mid\; s \Rightarrow t'\ \}}{L[t] \leadsto_L L[\mathit{fsm}\ \{\ v \;\mid\; s \Rightarrow t'\ \}]}$$

$$\begin{array}{rcl}
[\![\mathit{fsm}\ \{\ v \mid s \Rightarrow e_1\ \} * t_2]\!] & = & \mathit{fsm}\ \{\ v \mid s \Rightarrow \mathit{let}\ x = e_1\ \mathit{in}\ (x.1, x.2 * t_2)\ \} \\
[\![e_2 * \mathit{fsm}\ \{\ v \mid s \Rightarrow e_1\ \}]\!] & = & \mathit{fsm}\ \{\ v \mid s \Rightarrow \mathit{let}\ x = e_1\ \mathit{in}\ (x.1, e_2 * x.2)\ \} \\
[\![\mathit{fsm}\ \{\ v \mid s \Rightarrow e_1\ \} + t_2]\!] & = & \mathit{fsm}\ \{\ v \mid s \Rightarrow \mathit{let}\ x = e_1\ \mathit{in}\ (x.1, x.2 + t_2)\ \} \\
[\![e_2 + \mathit{fsm}\ \{\ v \mid s \Rightarrow e_1\ \}]\!] & = & \mathit{fsm}\ \{\ v \mid s \Rightarrow \mathit{let}\ x = e_1\ \mathit{in}\ (x.1, e_2 + x.2)\ \} \\
[\![!\ \mathit{fsm}\ \{\ v \mid s \Rightarrow e\ \}]\!] & = & \mathit{fsm}\ \{\ v \mid s \Rightarrow \mathit{let}\ x = e\ \mathit{in}\ (x.1, !x.2)\ \} \\
[\![\mathit{let}\ x = \mathit{fsm}\ \{\ v \mid s \Rightarrow e_1\ \}\ \mathit{in}\ t_2]\!] & = & \mathit{fsm}\ \{\ v \mid s \Rightarrow \mathit{let}\ s_1, x = e_1\ \mathit{in}\ (s_1, t_2)\ \} \\
[\![\mathit{let}\ x = e_1\ \mathit{in}\ \mathit{fsm}\ \{\ v \mid s \Rightarrow e_2\ \}]\!] & = & \mathit{fsm}\ \{\ v \mid s \Rightarrow \mathit{let}\ x = e_1\ \mathit{in}\ e_2\ \} \\
[\![\mathit{fsm}\ \{\ v \mid s \Rightarrow e\ \}.i]\!] & = & \mathit{fsm}\ \{\ v \mid s \Rightarrow \mathit{let}\ x = e\ \mathit{in}\ (x.1, x.2.i)\ \} \\
[\![(\bar{e}, \mathit{fsm}\ \{\ v \mid s \Rightarrow e\ \}, \bar{t})]\!] & = & \mathit{fsm}\ \{\ v \mid s \Rightarrow \mathit{let}\ x = e\ \mathit{in}\ (x.1, (\bar{e}, x.2, \bar{t}))\ \}
\end{array}$$

Figure 6. Lifting of nested FSMs.

$$M ::= [\cdot] \;\mid\; \mathit{fsm}\ \{\ v \;\mid\; s \Rightarrow M\ \}$$

$$\frac{[\![u]\!] = \mathit{fsm}\ \{\ v \;\mid\; s \Rightarrow e\ \}}{M[u] \leadsto_M M[\mathit{fsm}\ \{\ v \;\mid\; s \Rightarrow e\ \}]}$$

$$[\![\mathit{fsm}\ \{\ v_1 \mid s_1 \Rightarrow \mathit{fsm}\ \{\ v_2 \mid s_2 \Rightarrow e_2\ \}\ \}]\!] \quad = \quad \mathit{fsm}\ \{\ (v_1, v_2) \mid s \Rightarrow \mathit{let}\ s_1, s_2 = s\ \mathit{in}\ \mathit{let}\ x = e_2\ \mathit{in}\ ((x.2.1, x.1), x.2.2)\ \}$$

Figure 7. Merging of nested FSMs.

Truong and Hanrahan [2019] hold the view that it is a golden age for applying programming language techniques for improving hardware design. We believe implicit state machines may contribute to that initiative.

5 Implicit State Machines in Scala

To assess the feasibility of implicit state machines as a programming construct, we implemented an embedded DSL in Scala for digital design. We experimented usability of the embedded DSL by creating circuits of varying complexity, from half adders to a micro-controller.

5.1 Embedded DSL

For readers not familiar with DSLs, there are generally two approaches to implement a DSL:

- External DSL, in which the DSL is implemented with a standalone compiler (Figure 8)
- Embedded DSL, in which the DSL is defined as a library within a host language (Figure 9)

In the external approach, the language designer defines syntax of the DSL, users write DSL programs and then feed the source code into the DSL compiler. For practicality, there is the need to provide IDE support for the DSL to improve programming experience.

In the embedded approach, the language designer only needs to define the *abstract syntax tree* (AST) data format and provide core compiler phases as a library in an implementation language, e.g., Scala. Users write DSL programs in Scala to directly construct the ASTs, and then feed them into the compilation pipeline. As programmers write code in an existing language, e.g., Scala, there is no need to provide additional IDE support.

Given that the embedded approach avoids the overhead of defining concrete syntax of the DSL and providing IDE suport, we follow the approach in our work.

Our DSL is based on implicit state machines extended with pairs and bit vectors. Implicit state machines are the only state elements in the DSL. An excerpt of the abstract syntax tree definitions is presented in Figure 10.

The class `Sig` is the base class of abstract syntax trees, and the class `Type` is the base class of the types of signals. The DSL also defines the following aliases for types:

```
type ~[S <: Type, T <: Type] = PairT[S, T]
type Vec[T <: Num] = VecT[T]
type Bit = VecT[1]
type Num = Int
```

The type `Sig[Bit]` denotes signals of 1-bit vector, which is an alias of `Sig[Vec[1]]`. The type `Sig[Vec[2]]` denotes signals of 2-bit vector. Here we take advantage of *literal types* in Scala [Odersky 2019], which supports the usage of literal values as types.

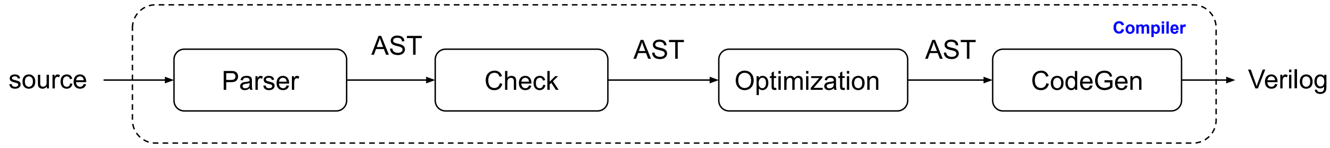


Figure 8. Architecture of External DSLs

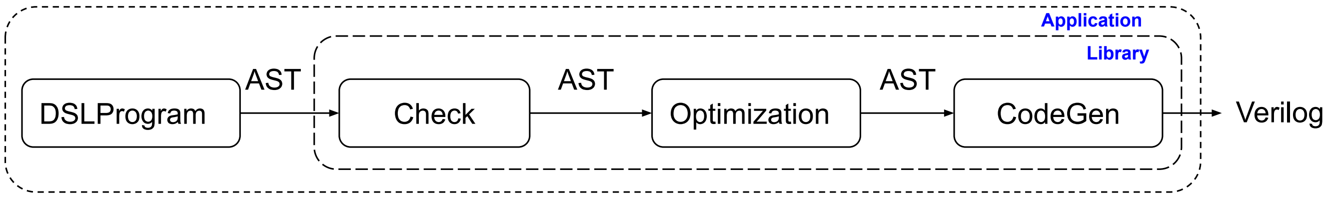


Figure 9. Architecture of Embedded DSLs

```scala
sealed abstract class Sig[T <: Type]    // base class of AST
case class Fsm[S <: Type, T <: Type](sym: Symbol, init: Value, body: Sig[S ~ T]) extends Sig[T]
case class Let[S <: Type, T <: Type](sym: Symbol, sig: Sig[S], body: Sig[T]) extends Sig[T]
case class Var[T <: Type](sym: Symbol, tpe: Type) extends Sig[T] // variable for inputs and bindings
case class And[T <: Num](lhs: Sig[Vec[T]], rhs: Sig[Vec[T]]) extends Sig[Vec[T]]
case class Mux[T <: Type](cond: Sig[Bit], thenp: Sig[T], elsep: Sig[T]) extends Sig[T]

sealed abstract class Type    // base class of types
case class PairT[S <: Type, T <: Type](lhs: S, rhs: T) extends Type
case class VecT[T <: Num](width: T) extends Type
```

Figure 10. An excerpt of abstract syntax trees of the DSL

The DSL supports common bit-wise operations such as XOR, AND, OR, ADD, SUB, SHIFT and MUX. All these operations are supported in Verilog [IEEE 2005], and they follow the same semantics as in Verilog.

The design intentionall makes the class `Sig` take an additional type parameter, which signifies the type of the signal. This way, we can profit the Scala type system to automatically check signal mismatch errors, e.g., perform OR operation on a 4-bit and an 8-bit signal.

5.2 A Quick Glance

The following code shows how we may implement a half adder in our DSL [2]:

```scala
def halfAdder(a: Sig[Bit], b: Sig[Bit]) =
  val s = a ^ b
  val c = a & b
  c ++ s
```

The operator ++ concatenates two bit vectors to form a bigger bit vector — `Sig[Vec[2]]` in the example above.

[2]For typesetting purposes, we omit some type annotations. For readability, we do not use *let/in* to manually optimize the circuit in places where the same sub-circuit is used multiple times.

We may compose two half adders to create a full adder, which takes a carry `cin` as input:

```scala
def full(a: Sig[Bit], b: Sig[Bit], cin: Sig[Bit]) =
  val ab = halfAdder(a, b)
  val s = halfAdder(ab(0), cin)
  val cout = ab(1) | s(1)
  cout ++ s(0)
```

In the above, we make two calls to `halfAdder`. Each call will create a copy of the half adder circuit to be composed in the fuller adder. It returns the carry and the sum. We may compose them further to create a 2-bit adder:

```scala
def adder2(a: Sig[Vec[2]], b: Sig[Vec[2]]) =
  val cs0 = full(a(0), b(0), 0)
  val cs1 = full(a(1), b(1), cs0(1))
  cs1(1) ++ cs1(0) ++ cs0(0)
```

To actually generate a representation of the circuit, we need to specify the input signals:

```scala
    val a = variable[Vec[2]]("a")
    val b = variable[Vec[2]]("b")
    val circuit = adder2(a, b)
```

Users may generate Verilog code for the circuit:

```scala
    circuit.toVerilog("Adder", a, b)
```

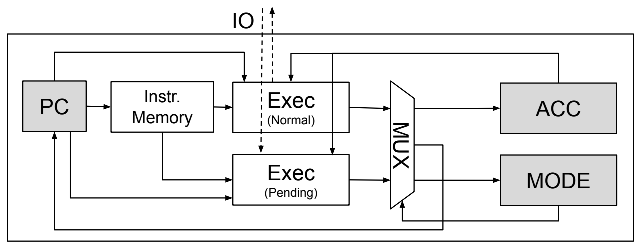


Figure 11. Architecture of the micro-controller

5.3 Sequential Circuits

We show how to create sequential circuits with the example of moving average filter. The moving average filter we are going to implement is specified below:

$$Y_i = (X_i + 2 * X_{i-1} + X_{i-2})/4$$

For the input X_i, the output Y_i also depends on the previous values X_{i-1} and X_{i-2}. We can define an operator `delay` based on implicit state machines:

```
def delay[T <: Type](sig: Sig[T], init: Value) =
  fsm("delay", init) { (last: Sig[T]) =>
    sig ~ last
  }
```

In the code above, we declare an implicit state machine with the specified initial state `init`. The body of the FSM is a pair `sig ~ last`, where the first part becomes the next state, and the second part becomes the output.

Now we may create the circuit for the moving average:

```
def movingAverage(in: Sig[Vec[8]]) =
  val z1 = delay(in, 0.toValue(8))
  val z2 = delay(z1, 0.toValue(8))
  (in + (z1 << 1) + z2) >> 2
```

In the code above, we first create an instance of the delay circuit and bind it to the variable `z1`. Then we delay the signal `z1` to get `z2`. Finally, the equation is encoded straightforwardly.

Note that in the above, the end user is programming in dataflow style à la Lustre [Caspi et al. 1987]. There is no need for the programmer to think in terms of state machines in such use cases. We discuss this in a broader context in Section 5.5.

5.4 Case Study: Micro-controller

We implemented an accumulator-based micro-controller in the DSL inspired by Leros [Schoeberl 2011].

The architecture of the micro-controller is shown in Figure 11. At the high-level, the micro-controller contains three architectural states: the program counter (PC), the accumulator register (ACC) and the pending status (MODE). The micro-controller interfaces with a memory bus, which contains a simple protocol consisting of read address, control (read / write) and data. The micro-controller contains an on-chip read-only instruction memory, which is different from the external data memory interfaced by the bus.

The micro-controller is implemented with an implicit state machine:

```
fsm("processor", pc0 ~ acc0 ~ mode0) { state =>
  val pc ~ acc ~ mode = state
  // ...
}
```

The variable `pc` refers to the program counter, `acc` is the accumulator register, `mode` indicates whether the controller is waiting for data from the external memory.

The skeleton of the implementation is as follows:

```
let("pcNext", pc + 1) { pcNext =>
  let("instr", readInstr(pc)) { instr =>
      /* ... */
      when (opcode === ADDI.toSig(8)) {
        val acc2 = acc + operand
        next(acc = acc2)
      }
      /* ... */
   }
}
```

It first increments the program counter `pc` and bind the result to `pcNext`. Then it binds the current instruction to `instr`. At the circuit-level, the operations are executed in parallel. Finally, the instruction is decoded and executed in a series of `when` constructs, depending on the mode of the micro-controller.

The `when` construct is a syntactic sugar created from multiplexers that supports selecting one of two n-bit inputs based a 1-bit control signal.

Eventually, each branch calls the local method `next` with appropriate arguments:

```
def next(pc: Sig[PC] = pcNext,
    acc: Sig[ACC] = acc,
    mode: Sig[Bit] = 0,
    out: Sig[BusOut] = 0)
= (pc ~ acc ~ mode) ~ out
```

As can be seen from above, the method `next` defines default values for all arguments, such that each branch may only specify parameters that are different. For example, the following code handles the unconditional jump instruction `BR`:

```
when (opcode === BR.toSig(8)) {
  next(pc = pc + jmpOffset)
}
```

The indirect `ADD` instruction needs to load data from external memory, thus putting the controller in the *pending* mode, as the following code shows:

```
when (opcode === ADD.toSig(8)) {
  next(pc = pc, mode = 1, out = readReq(instr))
}
```

The logic for the pending mode is as follows:

```
when (mode) {
  /* pending mode */
  when (opcode === ADD.toSig(8)) {
    next(acc = acc + busIn)
  }
  /* ... */
} otherwise {
  /* normal mode */
}
```

The code above depends on the protocol which requires that the I/O devices make the requested data available on the bus in the cycle following the request.

The programming experience is largely positive, thanks to the declarative nature of the DSL. Compared to VHDL or Verilog, there are no "wires" to connect in the DSL and there are no combinational cyles by construction.

We test the implementation with small assembly programs and verify the result with a circuit simulator in Scala. We are aware that the micro-controller is still too simple and it may not match quality standards. For example, we do not implement pipelining [Kroening and Paul 2001] nor do we separate out a reusable arithmetic-logic unit (ALU) for the two execution modes.

However, we conjecture that implicit state machines make it possible to automate some of such optimizations using compilation techniques. We leave it for future work to capitalize on such insights to implement RISC-V cores and compare with the state-of-the-art open source implementations.

5.5 Discussion: State Machine VS. Dataflow

It has long been observed that embedded systems fall into two categories: (1) control-dominated applications and (2) data-oriented applications [Colaço et al. 2005]. For control-dominated applications, programming based on finite-state machines is a good fit. For data-oriented applications, declarative dataflow programming is a good fit. However, real systems are usually a mix of both styles, which motivates the extension of the declarative dataflow language Lustre [Caspi et al. 1987] with state machines [Caspi and Pouzet 2008; Colaço et al. 2005]. The extension is in imperative style with explicit state representation, and it desugars to a core dataflow calculus.

Our work can be seen as taken an opposite approach to [Colaço et al. 2005]: Instead of desugaring finite-state machines into a core dataflow calculus, we make implicit state machines as the fundamental building block, and desugar dataflow programming constructs to implicit state machines (Section 5.3). Given that the dataflow calculus of Lustre eventually compiles to finite-state machines for execution, we believe the introduction of implicit state machines as a primitive will be an interesting addition to the programming methodologies of real-time and embedded systems.

6 Related Work

We have discussed related work in Section 4.5 and Section 5.5. Here we would like to acknowledge more work that inspired our research.

Our work is influenced by the french synchronous languages, Esterel [Berry and Gonthier 1992], Signal [Benveniste et al. 1991] and Lustre [Caspi et al. 1987]. In particular, the semantics of implicit state machines follow the *synchrony hypothesis* [Benveniste et al. 2003]. There are ongoing efforts in formalizing and mechanizing the semantics of these languages [Bourke et al. 2017; Florence et al. 2019] as well as verifying programs in these languages [Song and Chin 2021], which could be a direction for our future work.

There exists plenty of IRs for hardware design, such as Calyx [Nigam et al. 2021], FIRRTL [Izraelevitz et al. 2017], LLHD [Schuiki et al. 2020]. We believe implicit state machines will be a useful abstraction in the design of IRs due to its declarativeness, simplicity and universality.

There are many DSLs for digital design, including Lava [Bjesse et al. 1998; Gill et al. 2009], Chisel [Bachrach et al. 2012], Caisson [Li et al. 2011], Bluespec [Bourgeat et al. 2020], Kami [Choi et al. 2017], the Haskell-based DSL by Aronsson and Sheeran [2017]. Our DSL is different in the sense that it is based on a novel primitive, i.e., implicit state machines and the DSL is declarative.

Graphical representation of programs seems to be favored over text-based programs in some application domains. There are several visual languages for programming with finite-state machines, such as Statecharts [Harel 1987], SyncCharts [André and Peraldi-Frati 2000], Simulink/Stateflow [Hamon and Rushby 2007]. We are investigating how to combine the benefits of visualizal languages and text-based languages in programming embedded systems.

7 Conclusion

In this paper, we showed that by sticking to the design principle of declarativeness, we arrive at a novel abstraction: *implicit state machines*. Implicit state machines are recursively composable and universal, which makes them promising both as a programming model as well as intermediate representation.

We formalized the concept of implicit state machines in a calculus with Boolean algebra as the domain and showed that it serves as an elegant model of digital circuits.

We implemented an embedded DSL in Scala based on implicit state machines, which supports both dataflow style programming and state-machine style programming. We implemented a micro-controller in the DSL and the experience of programming with implicit state machines is positive.

Future Work. We are considering designing a standalone domain-specific language based on implicit state machines for the application domains of Internet of Things (IoT) and industrial automation.

References

Charles André and Marie-Agnès Peraldi-Frati. 2000. Behavioral Specification of a Circuit Using SyncCharts: A Case Study. In *26th EUROMICRO 2000 Conference, Informatics: Inventing the Future, 5-7 September 2000, Maastricht, The Netherlands*. IEEE Computer Society, 1091. https://doi.org/10.1109/EURMIC.2000.874620

Markus Aronsson and Mary Sheeran. 2017. Hardware software co-design in Haskell. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, Iavor S. Diatchki (Ed.). ACM, 162–173. https://doi.org/10.1145/3122955.3122970

Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. 2012. Chisel: Constructing hardware in a Scala embedded language. *DAC Design Automation Conference 2012* (2012), 1212–1221.

Hendrik Pieter Barendregt. 1985. *The lambda calculus - its syntax and semantics*. Studies in logic and the foundations of mathematics, Vol. 103. North-Holland.

A. Benveniste and G. Berry. 1991. The synchronous approach to reactive and real-time systems. *Proc. IEEE* 79, 9 (Sept. 1991). https://doi.org/10.1109/5.97297

Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (2003), 64–83. https://doi.org/10.1109/JPROC.2002.805826

Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. 1991. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming* 16, 2 (Sept. 1991). https://doi.org/10.1016/0167-6423(91)90001-E

Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.* 19, 2 (1992), 87–152. https://doi.org/10.1016/0167-6423(92)90005-V

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, 174–184. https://doi.org/10.1145/289423.289440

Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The essence of Bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 243–257. https://doi.org/10.1145/3385412.3385965

Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A Formally Verified Compiler for Lustre. (2017).

P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. 1987. LUSTRE: A Declarative Language for Real-time Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*. ACM, New York, NY, USA. https://doi.org/10.1145/41625.41641 event-place: Munich, West Germany.

Paul Caspi and Marc Pouzet. 2008. Synchronous Functional Programming : The Lucid Synchrone Experiment.

Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.* 1, ICFP (2017), 24:1–24:30. https://doi.org/10.1145/3110268

Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2005. A conservative extension of synchronous data-flow with state machines. In *Proceedings of the 5th ACM international conference on Embedded software - EMSOFT '05*. ACM Press, Jersey City, NJ, USA. https://doi.org/10.1145/1086228.1086261

Mary F. Fernández, Daniela Florescu, Alon Y. Halevy, and Dan Suciu. 2000. Declarative specification of Web sites with Strudel. *The VLDB Journal* 9 (2000), 38–55.

Spencer P. Florence, Shu-Hung You, Jesse A. Tov, and Robert Bruce Findler. 2019. A calculus for Esterel: if can, can. if no can, no can. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019). https://doi.org/10.1145/3290374

Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. 2009. Introducing Kansas Lava. In *Implementation and Application of Functional Languages - 21st International Symposium, IFL 2009, South Orange, NJ, USA, September 23-25, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6041)*, Marco T. Morazán and Sven-Bodo Scholz (Eds.). Springer, 18–35. https://doi.org/10.1007/978-3-642-16478-1_2

Nicolas Halbwachs, Daniel Pilaud, Farid Ouabdesselam, and Anne-Cecile Glory. 1989. Specifying, Programming and Verifying Real-Time Systems Using a Synchronous Declarative Language. In *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings (Lecture Notes in Computer Science, Vol. 407)*, Joseph Sifakis (Ed.). Springer, 213–231. https://doi.org/10.1007/3-540-52148-8_18

Grégoire Hamon and John M. Rushby. 2007. An operational semantics for Stateflow. *Int. J. Softw. Tools Technol. Transf.* 9, 5-6 (2007), 447–456. https://doi.org/10.1007/s10009-007-0049-7

Michael Hanus and Christof Kluß. 2009. Declarative Programming of User Interfaces. In *PADL*.

David Harel. 1987. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8, 3 (June 1987). https://doi.org/10.1016/0167-6423(87)90035-9

Timothy L. Hinrichs. 2011. Plato: A Compiler for Interactive Web Forms. In *PADL*.

IEEE. 2005. *IEEE Standard for Verilog Hardware Description Language*. IEEE.

Adam M. Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2017, Irvine, CA, USA, November 13-16, 2017*, Sri Parameswaran (Ed.). IEEE, 209–216. https://doi.org/10.1109/ICCAD.2017.8203780

Daniel Kroening and Wolfgang J. Paul. 2001. Automated Pipeline Design. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. ACM, 810–815. https://doi.org/10.1145/378239.379071

Michael Leuschel. 2008. Declarative programming for verification: lessons and outlook. In *PPDP '08*.

Xun Li, Mohit Tiwari, Jason Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: a hardware description language for secure information flow. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 109–120. https://doi.org/10.1145/1993498.1993512

Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A compiler infrastructure for accelerator generators. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 804–817. https://doi.org/10.1145/3445814.3446712

Martin Odersky. 2019. Scala Language Specification. https://scala-lang.org/files/archive/spec/2.13/.

Martin Schoeberl. 2011. Leros: A Tiny Microcontroller for FPGAs. In *International Conference on Field Programmable Logic and Applications, FPL 2011, September 5-7, Chania, Crete, Greece*. IEEE Computer Society, 10–14. https://doi.org/10.1109/FPL.2011.13

Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. LLHD: a multi-level intermediate representation for hardware description languages. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 258–271. https://doi.org/10.1145/3385412.3386024

Harald Søndergaard and Peter Sestoft. 1990. Referential transparency, definiteness and unfoldability. *Acta Informatica* 27 (1990), 505–517.

Yahui Song and Wei-Ngan Chin. 2021. A Synchronous Effects Logic for Temporal Verification of Pure Esterel. In *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12597)*, Fritz Henglein, Sharon Shoham, and Yakir Vizel (Eds.). Springer, 417–440. https://doi.org/10.1007/978-3-030-67067-2_19

Lenny Truong and Pat Hanrahan. 2019. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. In *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA (LIPIcs, Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:21. https://doi.org/10.4230/LIPIcs.SNAPL.2019.7