Configuration Synthesis for Programmable Analog Devices with Arco

Sara Achour  
MIT CSAIL, USA  
sachour@csail.mit.edu

Rahul Sarpeshkar  
Dartmouth College, MIT RLE, USA  
rahul.sarpeshkar@dartmouth.edu

Martin C. Rinard  
MIT CSAIL, USA  
rinard@csail.mit.edu

Abstract
Programmable analog devices have emerged as a powerful computing substrate for performing complex neuromorphic and cytomorphic computations. We present Arco, a new solver that, given a dynamical system specification in the form of a set of differential equations, generates physically realizable configurations for programmable analog devices that are algebraically equivalent to the specified system. On a set of benchmarks from the biological domain, Arco generates configurations with 35 to 534 connections and 28 to 326 components in 1 to 54 minutes.

Categories and Subject Descriptors D.3.4 [Processors]: Compilers; C.1.3 [Processor Styles]: Analog Computers
Keywords Compilers, Analog Computing, Languages

1. Introduction
Programmable analog devices have emerged as a powerful computing substrate for performing complex neuromorphic and cytomorphic computations [5, 7, 8, 31, 33, 34, 36, 37, 41]. These systems directly map state variables, transient variables, and parameters in the underlying scientific model to physical aspects of the circuit such as voltage and current.

Programmable analog devices target large biological dynamical systems such as gene–protein and neuron networks. Computations involving these networks are often used for medical dosage optimization, disease prediction, and understanding biological phenomena [23, 35]. The dynamics of biological systems are often oversimplified to make the computations tractable on digital systems, which reduces the accuracy of the model in relation to the corresponding physical system [6, 15, 40]. Studying the modeled dynamics on digital hardware is challenging since these systems are often stiff and therefore prone to numerical instability [13, 29]. In this context, a stiff system is a system that is prone to numerical instability unless the time steps taken are extremely small. Examples of stiff systems include biological systems with both fast and slow dynamics. Analog devices are excellent candidates for these computations:

• **Continuous Time Domain**: Analog devices operate in the continuous time domain, circumventing the time scale issues that often plague stiff dynamical system computations implemented with discretized time steps. Time does not “tick” in intervals as in standard clocked digital systems, but runs continuously and asynchronously.

• **Stochastic Behavior**: The noise present in these analog devices is directly analogous to the noise present in biological systems as they are mapped onto the hardware [33, 37]. This property promotes analog computations that accurately model the underlying biological phenomena.

• **Direct Mapping**: Since the dynamical system is directly mapped to components on the analog device, the maximum dynamical system size is directly proportional to the scale of the analog device.

Currently, there is no compiler support for these devices, so they are programmed manually. The result is a configuration that specifies which analog components to connect and which dynamical system variables and constants to assign to input and output components. The following properties complicate the automatic generation of these configurations:

• **Complex Analog Components**: The components in many analog devices implement complex dynamics and nontrivial relations such as transcendental functions [19, 28, 41]. Because the components are typically engineered for analog efficiency and generality rather than ease of use, there may be a substantial semantic gap between the dynamical system and the analog hardware.

• **Relation Entanglement**: The complex multifunctional components in our target analog hardware devices implement multiple interdependent relations. Mapping a differential equation onto even one such component can there-
fore activate a cascading sequence of interactions between the component and the remaining dynamical system variables.

- **Resource Constraints**: Analog devices typically provide a finite set of only partially connected components. Any successful mapping of the dynamical system onto the device must live within these constraints.

We present Arco, a solver which, given an analog device specification and dynamical system, synthesizes an algebraically equivalent configuration of the analog device:

- **Algebraic Unification**: Arco fully exploits complex analog components by using algebraic unification to find compact and efficient isomorphisms between the relations in the dynamical system and the provided hardware primitives.

- **Materialized Constants**: The Arco solver materializes new constants that configure and specialize general analog components so that they effectively implement the relations in the specified dynamical system.

- **Relation Entanglement**: Arco successfully manages cascaded relation entanglements by detecting and harmonizing otherwise potentially conflicting dynamical system and hardware variable implementations.

- **Resource Constraints**: Arco effectively works within the hardware resource constraints by tracking and exploiting (potentially partially) used hardware components and using an SMT solver to solve component interconnection problems.

Because Arco works with a hardware specification language that defines the capabilities of the underlying analog device, it can work with the full range of analog devices, including devices that provide powerful new analog primitives. The resulting analog hardware configurations are algebraically equivalent to the specified dynamical systems.

### 1.1 Contributions

We claim the following contributions:

- **Basic Approach**: We present a new approach for mapping dynamical systems that model complex biological phenomena onto programmable analog devices. Starting with a specification of the dynamical system, our approach automatically synthesizes an algebraically equivalent configuration of the programmable analog device.

- **Analog Circuit Solver**: We present a new analog circuit solver that uses a deductive algebraic approach to solve the analog circuit synthesis problem. Algebraic unification enables Arco to fully exploit the powerful but complex analog components that modern programmable analog devices offer. Arco successfully manages cascading relation entanglements, materializes constants that appropriately configure general analog components, and works within the resource constraints of the programmable analog device.

- **Evaluation**: We evaluate Arco on five biological dynamical systems extracted from an artifact repository for biological models that have been accepted into computational biology conferences [1]. Given a programmable analog hardware description, Arco synthesizes configurations for the models with from 28 to 326 components and 35 to 534 connections in 1 to 54 minutes.

To enable these contributions, Arco (1) works with analog device descriptions written in a novel hardware specification language and (2) accepts dynamical systems written in a novel high level dynamical system specification language. To the best of our knowledge, Arco is the first system to automatically synthesize programmable analog device configurations from dynamical system specifications.

### 2. Example

We next present an example that demonstrates (1) how to express a dynamical system in the dynamical system specification language, (2) how to define a programmable analog device in the hardware specification language, and (3) the resulting solution generated by the solver.

#### 2.1 Dynamical System: Enzyme-Substrate Reaction

Consider a chemical reaction where an enzyme (E) reacts with a substrate (S) to produce a compound (ES), where the forward reaction rate is catalyzed by catalyst (Q). The formation rate of ES \( k_f \) is the sum of some nominal rate \( k_0 \) and the amount of the catalyst present multiplied by its efficiency constant \( k_r \), which we assume to be one. We therefore express the reaction rate for producing ES as \( k_0 + Q \).

The reverse reaction rate is \( k_r \).

\[
E + S \rightarrow_{k_0 + Q} ES \\
ES \rightarrow_{k_r} E + S
\]

We may express this reaction as a dynamical system comprised of differential equations, where \( E_{\text{tot}} \) and \( S_{\text{tot}} \) are the total amounts of E and S, respectively. The following set of equations specifies the behavior of the dynamical system:

\[
\begin{align*}
E &= E_{\text{tot}} - ES \\
S &= S_{\text{tot}} - ES \\
\frac{dES}{dt} &= (Q + k_0) \cdot E \cdot S - k_r \cdot ES
\end{align*}
\]

Figure 1 presents the differential equations written in the Arco dynamical system specification language (DSSL). The specification defines two units, seconds (s, unit of time) and molarity (\( \text{M} \), unit of concentration). The compound Q is expressed as an input quantity with unit of measure \( \text{M} \) (the quantity is measured as a concentration). The compounds E and S are expressed as intermediate (\text{local}) variables also measured in molarity M. The reaction parameters \( k_0 \) and \( k_r \) are assigned appropriate values and units. The time variable is \( t \) measured in seconds (s).

The relation statements (\text{rels}) describe the dynamics of each \text{local} and output variable. These statements implement the set of equations outlined above. The \text{init} clause in the definition of ES sets the initial value of ES to 0.423.
2.2 Hardware Specification

Figure 2 presents the hardware specification for a simple programmable analog device. The specification is inspired by existing and envisioned programmable analog chips that Arco is designed to target [8, 37, 41]. The specification defines microseconds (μs), milliseconds (mA), and millivolts (V) as units then maps time (t) to microseconds, the current property (I) to milliamps, and the voltage property (V) to millivolts.

The specification defines three components, a current adder (iadd), a current multiplier (igain) and a Michaelis-Menten component (mm) [37]. The current adder has two input ports (W and U) and one output port (Y). The current at the output port is the sum of the currents of the two input ports. The Michaelis-Menten component has five input ports A0, B0, Kf, Kr, and C0 and three output ports A, B, and C. The component simulates a basic enzyme-substrate reaction of the form $A + B \rightarrow C$:

$$A_V = B0_V - C_V \quad B_V = B0_V - C_V$$

The specification also defines available connections. The conn iadd->mm statement, for example, states that output ports of iadd components can be connected to input ports of mm components. A * specifies either all output (* ->) or input (-> *) ports of all components.

2.3 Arco Solver

The Arco solver works with a set of goals. Initially the goals are simply the relations in the dynamical system. As the solver operates it maps computations within the relations onto the hardware components, with the dynamical system variables and expressions mapped to properties (voltage, current) of the component ports. In our example the solver starts with the following goal table:

$$S = S_{tot} - ES \quad E = E_{tot} - ES$$

The solver first chooses to map the $S = S_{tot} - ES$ computation onto one of the Michaelis-Menten (mm) hardware components. The initial step is to map the dynamical system variable S onto the hardware variable Av. Each hardware variable is the combination of a port (here A) and a property

```plaintext
type s : type M
input Q : 1/(s*M)
local E : M; local S : M; output ES : M
param ETOT : M = 0.15; param STOT : M = 0.11
param k0 : 1/(s*M) = 0.36; param kr : 1/s = 0.2
time t : s
rel S = STOT - ES
rel E = ETOT - ES
rel deriv(ES,t)=((k0 + Q)*E*S)-kr*ES init 0.423
```

Figure 1. Enzyme Substrate Reaction Dynamical System

```plaintext
Figure 2. Analog Hardware Specification
```

such as voltage (V) or current (I). So $A_V$ is the voltage V property (in black) of the output port A:

$$A_V = A0_V - C_V \quad S = S_{tot} - ES$$

Arco then algebraically unifies the two relations to obtain the following assignments from hardware variables to dynamical system variables and expressions:

$$A_V = S \quad A0_V = S_{tot} \quad C_V = ES$$
In effect, the unification maps variables and expressions from the dynamical system onto the hardware variables of the mm component. In this case, the unification maps $S_{tot}$ onto $A0_V$ (the voltage property of port A0) and $ES$ onto $C_V$ (the voltage property of port C):

At this point Arco again encounters relation entanglement: it has mapped $E$ to an output port property, specifically $B_V$. It has two potentially conflicting definitions of $E$, one from the hardware and one from the dynamical system. It applies the current set of assignments to the hardware relation that defines $B_V$ ($B_V = B0_V - C_V$) to obtain the following mixed-variable relation that the hardware implements to compute $E$:

$$E = B0_V - ES$$

Arco retrieves the definition of $E$ from the dynamical system ($E = E_{tot} - ES$) and harmonizes the two relations with another algebraic unification. The unification maps $E_{tot}$ to $B0_V$ as follows:

Arco has now successfully mapped the original goal ($S = S_{tot} - ES$) onto the mm component, including performing any cascaded mappings required to harmonize any generated potential conflicts. Because of these cascaded mappings, Arco has partially mapped all of the following dynamical system relations onto the hardware:

$$S = S_{tot} - ES \quad E = E_{tot} - ES \quad \partial ES/\partial t = (k_0 + Q) \cdot S \cdot ES$$

The partial mapping generated a set of subgoals, which consist mostly of assignments of dynamical system variables to (analog) hardware variables. Arco therefore removes the partially mapped original goals and adds the generated subgoals to obtain the following goal table:

With the exception of the $Kf_1$ goal, the new goal table contains only assignments of dynamical system variables to hardware variables. Arco assumes that the analog hardware device will be embedded within a larger digital computing framework that will supply dynamical system input variables and values as digital numbers. The framework will receive dynamical system output variables as a sequence of digital values derived by sampling the analog representation of these variables at a specific frequency. The analog hardware device therefore contains input components, which use a digital to analog converter to import a digital input value into the analog computation, and output components, which use an analog to digital converter to produce the sampled digital sequence of output variable values.
Arco solves all of the goals that assign dynamical system input or output variables or values to analog hardware variables by inserting appropriate input or output components to import or export the relevant values to or from the analog computation:

At this point the goal table contains only one goal:

\[ K_f = Q + k_0 \]

Arco solves this goal by mapping the \( Q + k_0 \) computation onto an iadd hardware component, then connecting the iadd output port to the \( K_f \) input port of the \( mm \) component. It then uses appropriate input components to solve the remaining goals, which assign the \( Q \) and \( k_0 \) variables to iadd hardware input variables. Unlike the previous collection of inputs, the iadd requires current, not voltage, analog inputs. Arco therefore selects input components that produce current, not voltage, values.

At this point Arco has successfully solved all of the goals in the goal table. The result is an analog hardware configuration that correctly implements the specified dynamical system. The configuration specifies the generated inputs, outputs, and analog hardware connections:

2.4 Arco Search Space and Tableau Configurations

In practice, the Arco solver typically has many potential options at each step as it maps the dynamical system onto the hardware. Not all of these options lead to a successful hardware configuration. Arco therefore deploys a search algorithm to explore a search space of partial hardware configurations. The algorithm uses a variety of optimizations and heuristics to make the search tractable (Section 4.3).

Arco organizes the search with a tableau (Section 4.1). This data structure tracks the goals, partially used hardware, and generated partial hardware configurations. The search starts with a tableau configuration containing the analog hardware and the dynamical system. It generates a space of tableau configurations by mapping dynamical system variables, expressions, and relations onto the analog hardware components. The search of the generated tableau space terminates when Arco finds a solved tableau configuration with an empty goal table (Section 4.2).

3. Languages

Arco works with (1) a dynamical system specification that describes the dynamical system to model and (2) a hardware specification that describes the target programmable analog device.

3.1 Dynamical System Specification Language

The dynamical system specification language (DSSL) allows the programmer to describe the dynamical system to simulate. Figure 3 presents the syntax for the DSSL.

- **Unit**: A unit (type \( x \)) is a named unit of measurement that may be composed into unit expressions. Inputs, outputs, and parameters are all declared with unit expressions that define the units in which they are measured.
- **Time**: The time \((\text{time } x : x')\) is a named time variable \(x\) with a unit expression \(x'\) in which time is measured.
- **Variables**: A variable (**input**, **output**, **local** \( x : U \)) is a named quantity \(x\) with an attached unit expression \(U\).
- **Parameter**: A parameter (**param** \( x : U = f \)) is set to a fixed floating point value \(f\) that does not change over time.
- **Relations**: A relation \((\text{rel } x=E, \text{rel deriv}(x,x')=E \text{ init } f)\) defines the relationship between a local or output variable and the other variables in the dynamical system.

**Type Checking Rules**: Appendix A presents type checking rules for the dynamical system specification language.

3.2 Analog Hardware Specification Language

Figure 4 presents the syntax of the hardware specification language. Specifications contain named component definitions, which specify the computational primitives of the programmable analog device. These primitives are defined as a set of relations over properties (such as current and voltage) of the input and output ports of the component.
Units: A unit (type $x$) is a named unit of measurement for a specific property of the circuit or time. Units relate circuit properties to computational properties.

Time: The time (time $x : x'$) is a named time variable $x$ and the associated units $x'$ in which it is measured.

Properties: A property (prop $x : x'$), is a named circuit property $x$, such as current or voltage, and the associated unit $x'$ in which it is measured. The combination of a property $x$ and a port $x''$, written $x(x'')$, is a hardware variable.

Components: Components (comp $x$ ... end block) are the basic computational primitives of the circuit. Components have input ports (input $x$), output ports (output $x$), and parameters (param $x : x''$). Stateless relations $rel$ $Q=E$ define the value of an output port property $Q$ as an expression $E$. Stateful relations $rel$ deriv($Q(x)$)=$E$ init $Q'$, define the value of output port property $Q$ with a differential equation. $Q'$ provides the initial value.

Arco also supports input components, which provide a digital interface for importing dynamical system values into the analog circuit, and digital output components, which provide a digital interface for exporting dynamical system values from the analog circuit to the surrounding digital computing environment.

Schematic: A schematic end description block specifies resources $R$, including the number of instances of each component $inst$ $x : n$ and the potential connections between components $inst$ $x : n$ (at the granularity of components, instances of components, or individual component ports).

Type Checking Rules: Appendix B presents the type checking rules for the hardware specification language.

4. Arco Solver

Dynamical System: The Arco solver works with a dynamical system with input variables $\vec{\bar{i}} \in \bar{I}$, output variables $\vec{\bar{o}} \in \bar{O}$, and local variables $\vec{\bar{l}} \in \bar{L}$. Together these variables are the dynamical system variables $\vec{\bar{v}} = \bar{V} = \bar{I} \cup \bar{O} \cup \bar{L}$. It also works with sets of dynamical system relations $\bar{F} \in \bar{R}$, where each relation $\bar{r}$ is of the form $\bar{r}: \vec{\bar{v}} = \bar{E}$ or $\bar{r}: \partial \vec{\bar{v}}/\partial t = \bar{E}$ init $x$, $\vec{\bar{v}} \in \bar{V} \cup \bar{O}$, $x \in \bar{R}$, and $\bar{E}$ is an expression over $\bar{V}$ so that $vars(\bar{E}) \subseteq \bar{V}$.

Hardware: The hardware has digital to analog (DAC) input components $\bar{\bar{c}} \in \bar{\bar{C}}$, analog to digital (ADC) output components $\bar{\bar{c}} \in \bar{\bar{C}}$, and analog computation components $\bar{\bar{c}} \in \bar{\bar{C}}$. Together these components are the hardware components $\bar{\bar{c}} \in \bar{\bar{C}}$. The hardware specification language enables the designer to specify multiple instances of declared components. Each such instance is a separate component $\bar{\bar{c}} \in \bar{\bar{C}}$.

Components have input ports $\bar{\bar{i}} \in \bar{\bar{I}}$ and output ports $\bar{\bar{o}} \in \bar{\bar{O}}$. Together these ports are the ports $\bar{\bar{p}} \in \bar{\bar{P}} = \bar{\bar{I}} \cup \bar{\bar{O}}$. Each port $\bar{\bar{p}}$ belongs to a component $\bar{\bar{c}}$, denoted $\bar{\bar{p}} \in \bar{\bar{C}}$. Each port $\bar{\bar{p}}$ has properties $q \in \bar{Q}$, denoted $\bar{\bar{p}}$. Common properties $q$ include current and voltage.

The combination of a port and a property is a hardware variable. There are input variables $\bar{\bar{v}}_q \in \bar{\bar{V}} = \bar{\bar{I}} \cup \bar{\bar{O}}$ and output variables $\bar{\bar{v}} \in \bar{\bar{V}}$. Together these variables are the hardware variables $\bar{\bar{v}} = \bar{\bar{V}} = \bar{\bar{I}} \cup \bar{\bar{O}}$. Each variable $\bar{\bar{v}}$ is a property of a port $\bar{\bar{p}}$, denoted $\bar{\bar{v}} = \bar{i} \bar{\bar{p}}$.

Each component $\bar{\bar{c}} \in \bar{\bar{C}}$ has a set of variables $vars(\bar{\bar{c}}) \subseteq \bar{\bar{V}}$. Components have disjoint variables so that $\bar{\bar{c}}_1 \cap \bar{\bar{c}}_2 = \emptyset$. Input components $\bar{\bar{c}} \in \bar{\bar{C}}$ have a single digital input variable $\bar{i}_q$ (the input port $\bar{i}$ with the digital property $d$). Output components $\bar{\bar{c}} \in \bar{\bar{C}}$ have a single digital output variable $\bar{o}_q$ (the output port $\bar{\bar{o}}$ with the digital property $d$).

The hardware implements a set of hardware relations $\bar{\bar{r}} \in \bar{\bar{R}}$, where each relation $\bar{\bar{r}}$ is of the form $\bar{\bar{r}}: \vec{\bar{v}} = \bar{E}$ or $\bar{\bar{r}}: \partial \vec{\bar{v}}/\partial t = \bar{E}$ init $\bar{\bar{v}}$, where $\vec{\bar{v}} \in \bar{\bar{O}}$ and $\bar{E}$ is an expression over $\bar{\bar{V}}$ so that $vars(\bar{E}) \subseteq \bar{\bar{V}}$.

The hardware also has a set of wires $\bar{\bar{w}} \subseteq \bar{\bar{I}} \cup \bar{\bar{O}}$. Each wire is a configurable connection between an output port and an input port.

Goals: As the solver operates, it maps the dynamical system relations onto the hardware (by mapping variables and expressions from the dynamical system to properties of hardware ports). The solver works with a set of goals $\bar{\bar{F}} \in \bar{\bar{R}}$, each of which is derived from an original dynamical system relation $\bar{\bar{r}} \in \bar{\bar{R}}$ and each of which captures some remaining unmapped part of $\bar{\bar{r}}$. The solver terminates when all of the relations in the dynamical system have been fully mapped onto the hardware and the set of goals is empty.
solve(\(\mathbb{R}, W, \mathbb{R}\)) = \{Z | \langle \mathbb{R}, 0, W, \mathbb{R}, 0 \rangle \to * \langle \mathbb{R}', R', W', 0, Z \rangle\}

\[
\text{UNIFY} \\
\begin{array}{c}
\forall r \in \mathbb{R} \cup \mathbb{R} \\
\exists \bar{r} \in \mathbb{R} \\
\text{unify}(r, \bar{r}, \mathbb{R}, \mathbb{R}) = (\mathbb{R}', \mathbb{R}')
\end{array}
\]

\[
\text{CONNECT} \\
\begin{array}{c}
\exists I_q \in \mathbb{R} \\
\exists w : (\sigma, I) \in W \\
\langle \mathbb{R}, W, I, \mathbb{R} \rangle \to \langle \mathbb{R}, W - \{w\}, \mathbb{R} - \{I\}, Z \cup \{I \mapsto I_q\} \rangle
\end{array}
\]

\[
\text{INPUT-VAL-MAP} \\
\begin{array}{c}
\exists I_d = x \in R \\
\exists I \in I \\
I \mapsto I \cup \{x \mapsto I_d\}
\end{array}
\]

\[
\text{OUTPUT-VAL-MAP} \\
\begin{array}{c}
\exists I \in I \\
\exists z \in Z
\end{array}
\]

\[
\begin{array}{c}
\exists I_d = I_q \in \mathbb{R} \\
\exists x \in \mathbb{R} \\
\exists I \in I \\
I \mapsto I \cup \{x \mapsto I_d\}
\end{array}
\]

Each goal \(\bar{r} \in \mathbb{R}\) is a relation \(\bar{r}\) of the form \(\bar{r} : \hat{\nu} = \hat{E}\), \(\bar{r} : \frac{\partial E}{\partial \bar{r}} = \hat{E}\) init \(x\) (relations from the dynamical system), \(\bar{r} : \hat{I} = E\) (a mixed-variable derived relation that may include both dynamical system and hardware variables), or \(\bar{r} : \hat{\nu} \mapsto \bar{q}\) (mapping a dynamical system local or output variable to a hardware output variable), where \(\hat{v} \in L \cup \hat{O}\), \(x \in \mathbb{R}\), \(\hat{E}\) is an expression over \(\hat{V}\) so that \(\text{vars}(\hat{E}) \subseteq \hat{V}\), and \(E\) is an expression over \(V = \hat{V} \cup \hat{V}\) so that \(\text{vars}(E) \subseteq V = \hat{V} \cup \hat{V}\), \(\bar{I}_q \in \bar{I}\), and \(\bar{q} \in \bar{Q}\).

Note the solver may generate partially solved mixed-variable goals that include both dynamical system and hardware variables.

**Partially Used Hardware:** As the solver operates, it consumes the starting hardware \(\hat{r} \in \hat{R}\) to produce partially used hardware \(\hat{r} \in \hat{R}\). Each relation \(\hat{r} \in \hat{R}\) is of the form \(\hat{r} : \hat{I} = E\) or \(\hat{r} : \hat{I} = \hat{E}\) init \(\hat{I}_q\), where \(\text{vars}(E) \subseteq V = \hat{V} \cup \hat{V}\). Each \(r \in \hat{R}\) is therefore a partially instantiated mixed-variable relation derived from an original hardware relation.

**Hardware Configuration:** As it operates, the solver generates a hardware configuration \(Z\). Each \(z \in Z\) is of the form:

- \(\bar{I} \mapsto \hat{I}\): Connect a wire from hardware output port \(\hat{I}\) to hardware input port \(\bar{I}\).
- \(x \mapsto I_d\): Set a digital input \(I_d \in \bar{I}\) to a value \(x \in \mathbb{R}\).
- \(\hat{r} \mapsto I_d\): Map a dynamical system input variable \(\hat{r}\) to a digital input \(I_d \in \bar{I}\).
- \(\hat{v} \mapsto \bar{q}\): Map a dynamical system output or local variable \(\hat{v} \in L \cup \hat{O}\) to a hardware output port property \(\bar{q} \in \bar{Q}\).

which contains the initial unused hardware environment \(\hat{R}\), wires \(W\), and dynamical system relations \(\mathbb{R}\) as goals.

Figure 5 presents the rules that define the solver steps. Rule unify maps a (potentially already partially mapped) relation \(\bar{r} \in \mathbb{R}\) onto a hardware relation \(\hat{r} \in \hat{R}\). Rules connect through output-\-var-map solve goals that directly configure the hardware:

- **connect:** The connect rule solves a goal of the form \(\bar{I} = \bar{I}_q\) by adding a wire \(\bar{I} \mapsto \bar{I}_q\) to the configuration \(Z\). The wire connects the output port \(\bar{I}\) to the input port \(\bar{I}^\prime\).
- **input-var-map:** The input port mapping rules solve goals of the form \(\bar{I}_q = x\) or \(\bar{I}_q = \bar{I}\). The rules add the mapping \(x \mapsto \bar{I}_q\) or \(\bar{I} \mapsto \bar{I}_q\) to the configuration \(Z\). The mapping binds the value \(x\) or dynamical system input \(\bar{I}\) to the hardware variable \(\bar{I}_q\).
- **local-var-map:** The local variable mapping rule solves goals of the form \(\hat{I} \mapsto \bar{q}\). It adds the mapping to the configuration \(Z\) and adds \(\bar{I}_q = \hat{I}\) to \(\hat{R}\). Arco solves resulting future subgoals of the form \(\bar{I}_q = \hat{I}\) through unification, provided \(\bar{I}_q = \bar{q}\) holds.
- **output-var-map:** The output variable mapping rule solves goals of the form \(\bar{I} \mapsto \bar{q}\). The rule uses a digital output component \(\bar{I}_d = \bar{I}_q\). It also adds the mapping \(\bar{I} \mapsto \bar{I}_d\) to \(X\), the relation \(\bar{I}_d = \hat{I}\) to \(\hat{R}\), and the goal \(\hat{I}_q = \bar{I}_d\) to \(\hat{R}\).

The unify rule uses the unify rule that defines the unify function. Each rule first unifies \(r\) and \(\bar{r}\) and resolves any entangled and affected relations. Figure 6 presents the transition rules that define the unify function. Each rule first unifies \(r\) and \(\bar{r}\) to obtain a set of assignments \(A\). It then applies the assignments \(A\) to \(\hat{R}\) (if \(r \in \hat{R}\)) or to \(\hat{R}\) if \(r \in R\).

The solver steps unify relations \(r\). The basic Arco unification algorithm  

\(A = \text{unifyExp}(E, E')\) operates on expressions \(E\) and \(E'\). It produces sets of assignments \(A\) that unify \(E\) and \(E'\) so that \(\text{sub}(E, A) = \text{sub}(E', A)\), where \(\text{sub}(E, A)\) applies the assignments \(A\) to the expression \(E\).

The rules in Figure 7 lift the unification algorithm from expressions \(E\) to relations \(r\).
Relation Entanglement: Unifying a partially used \((r \in \bar{R})\) or unused \((r \in \bar{R})\) hardware relation with some goal \(\bar{T} \in R\) (where the variables \(v \in V\) are the hardware variables in \(\bar{T}: r \oplus \Box\)) may produce assignments which are entangled with other relations that belong to \(\Box\) in \(R\). A hardware relation is entangled with an assignment \(a: \Box q = E\) if the relation defines \(\Box q\) (for example, \(\Box q = E\)). A hardware relation is affected by \(a\) if \(\Box q\) is used (but not defined) by the relation (for example, \(\Box q = E\) where \(\Box q \in \text{vars}(E)\)).

Detangling Relations: Figure 8 presents the transition relations that apply a set of assignments \(A\) to the tableau. The rules (1) detach any relations entangled with \(A\) \((\rightarrow_d)\) and (2) move any relations affected by \(A\) out of \(R\) \((\rightarrow_o)\). Harmonization detangles entangled relations. Given an assignment \(\Box q = E\) and a relation \(\Box q = E'\), Arco first applies the assignments \(A\) to \(E'\) to create the specialized expression \(E'' = \text{sub}(E', A)\). It unifies expressions \(E\) and \(E''\) (assigning only variables \(V\)), then incorporates the resulting set of assignments \(A'\) (chosen from the space of possible assignments) into \(A\).

The \(\rightarrow_d\) relation detangles \(r \in R\) from \(A\) by incrementally applying assignments \(a \in A\) to the tableau and detangling any entangled relations as needed, where \(\bar{A}\) are the remaining assignments. The rules move detangled relations into \(D\):

- **APPLY-IN-ASSIGN:** Assignments of the form \(\Box q = E\) create no entanglements. The rule therefore moves such assignments directly into the goal set \(\bar{R}\).
- **APPLY-OUT-ASSIGN:** The rule harmonizes entangled assignments of the form \(a: \Box q = E\) where \(r: \Box q = E' \in R\). It then adds the resulting set of assignments \(A'\) to \(\bar{A}\) and
adds the fully specialized relation \( \overline{\sigma}_q = \text{sub}(E', A \cup A') = E \) to \( D \). Because the harmonization satisfies assignment \( a \), the rule does not add \( a \) to the goal set \( \overline{R} \).

- **APPLY-OUT-ASSIGN:** The rule harmonizes entangled assignments of the form \( a : \overline{\sigma}_q = \overline{v} \) where there is an entangled \( \overline{r} : \overline{\sigma}_q = E' \in R \), and a definition of \( \overline{v} \) of the form \( \overline{t} : \overline{v} = \overline{E} \in \overline{R} \). The rule harmonizes \( \overline{E} \) and \( E' \) (with \( V \)). The rule makes the same changes to \( A \), \( A \), and \( D \) as APPLY-OUT-ASSIGN, but also removes the now-mapped relation \( \overline{t} \) from the goal set \( \overline{R} \).

**Resolving Affected Relations:** The \( \to_r \) relation resolves any affected relations \( r \in R \) after all potentially entangled relations have been resolved (when \( \overline{A} = \emptyset \)). Each \( \to_r \) rule finds a relation \( r \in R \) affected by the assignments \( A \). It then applies the assignments \( A \) to \( r \) to create a specialized relation which it adds to \( D \). It then removes \( r \) from \( R \). It applies these transitions until no affected relations are left.

### 4.2 The Search Algorithm

Figure 9 presents the Arco search algorithm. The algorithm explores the search space defined by the tableau configuration transition relation \( \to \). It maintains a set of tableau configurations \( F \) at the frontier of the explored space. At each step, the algorithm chooses a tableau configuration \( \overline{t} : (\overline{R}, \overline{R}, \overline{W}, \overline{R}, Z) \in F \) to explore. If all of the goals in \( \overline{t} \) have been solved (i.e., \( \overline{R} = \emptyset \)), the algorithm returns the generated configuration \( Z \). Otherwise, it selects a subset \( T \) of the set of tableau configurations directly reachable from \( \overline{t} \) under \( \to \) and replaces \( \overline{t} \) in \( F \) with \( T \). The decisions that Arco makes when it chooses a tableau configuration \( \overline{t} \) (line 3) and selects a subset \( T \) to explore (line 5) have a significant impact on the effectiveness of the search algorithm.

**Choosing \( \overline{t} \):** At line 3, Arco applies a heuristic designed to find the tableau configuration \( \overline{t} \) that is closest to being solved. This heuristic scores each tableau configuration based on the number and complexity of the remaining nontrivial goals (goals solved only by rule \( \text{UNIFY} \) in Figure 5). It chooses the configuration \( \overline{t} \) with the smallest score (i.e., fewest and simplest remaining nontrivial goals).

**Selecting \( T \):** Arco first selects a goal \( \overline{r} \in \overline{R} \), prioritizing trivial goals (solved by rules \( \text{connect} \) through \( \text{OUTPUT-VAR-MAP} \) in Figure 5) over nontrivial goals (solved by rule \( \text{UNIFY} \) in Figure 5). It then computes the set of tableau configurations generated by applying the appropriate rule from Figure 5 to \( \overline{t} \) to obtain \( T \). If any of the trivial goals are unsolvable, Arco sets \( T = \emptyset \), effectively pruning the entire search subspace rooted at \( t \). The rationale is that the unsolvable trivial goal ensures that the pruned subspace contains no solved tableau configurations.

The \( \text{UNIFY} \) rule from Figure 5 may generate a large number of tableau configurations. To maintain the tractability of the search algorithm, the current Arco implementation discards generated tableau configurations so that \( T \) contains at most five tableau configurations from each combination of \( \overline{t} \) and \( \overline{r} \) in the \( \text{UNIFY} \) rule.
Search Tree Data Structure: Instead of fully storing each tableau configuration in the frontier \( F \), Arco maintains (1) a full representation of the current tableau configuration and (2) a delta-based representation of the explored tableau search space. Arco navigates the space by applying and unapplying deltas to the current tableau configuration. This optimization significantly reduces the space required to explore the search space.

4.4 Implementation Details

Arco is implemented in OCaml. Arco includes a collection of utility libraries, including OCaml bindings for sympy [22], a python library for algebraic simplification, and OCaml bindings for the Z3 SMT solver [9]. It also includes implementations of several custom data structures and convenience routines.

4.5 Design Decisions

We decided to use algebraic unification rather than pattern-based unification because the analog circuit synthesis problem requires more sophisticated reasoning than pattern-based unification systems can deliver. We found that synthesizing configurations for programmable analog devices often hinges on the solver’s ability to apply algebraic reasoning to rewrite algebraic expressions during unification.

We found algebraic reasoning to be particularly effective at materializing the new constants required to appropriately configure general analog components and at algebraically transforming dynamical system computations to exploit efficient analog components with complex interfaces. This kind of algebraic reasoning is beyond the capabilities of pattern-based unification and other methods that pattern match expression trees. For this reason, we used a computer algebra system that came with a corpus of axioms and was already effective at algebraically expressing trees. For this reason, we used a computer algebra system that came with a corpus of axioms and was already able to reason effectively about algebraic expressions.

We chose to build a deductive solver instead of encoding the synthesis problem as an SMT problem because SMT techniques are not well suited for reasoning about continuous, transcendental, and nonlinear functions. We found that attempting to encode algebraic reasoning directly in SMT quickly became intractable. Utilizing a deductive technique allowed the solver to synthesize solutions in a more focused manner than an exhaustive search over valid configurations. We use an SMT solver for more appropriate subproblems such as the concretization of abstract configurations.

We implemented a specialized procedure that performs relation detanglement and affected relation resolution after observing that the initial relation unification often produced potential conflicts between the definitions of dynamical system variables and corresponding entangled analog hardware computations. Immediate harmonization reduces the size of the search space and allows Arco to use simpler tableau transitions.

5. Experimental Results

We present experimental results for Arco on a set of benchmarks of varying size and complexity, given a hardware model with components commonly found in programmable analog devices that target biological computations. The benchmarks are a selection of published artifacts from well-cited computational biology papers [1]. The benchmarks were automatically converted into the dynamical system specification language from the provided Octave files. We use the following benchmarks (Table 1):

- **Michaelis-Menten:** The Michaelis-Menten system models a reaction of the form: \( E + S \rightarrow ES \rightarrow P \) [30]. The corresponding dynamical system has 9 parameters, 2 inputs, 3 functions, and 2 differential equations.
- **Genetic Toggle Switch for E. coli:** The genetic toggle switch is a bistable gene-regulatory network found in E. coli. It consists of two repressible promoters [18]. The corresponding dynamical system has 3 parameters, 1 input, and 4 differential equations.
- **Transcriptional Reprissilator:** The LacI-tetR-CI transcriptional reprissilator system is an oscillating network composed of genes that are not involved in maintaining a biological clock [11]. The corresponding dynamical system has 7 parameters, 3 functions, and 2 differential equations.
- **Circadian Oscillator:** The circadian oscillator system is a minimal model for circadian oscillations based on a mutually interacting activator and repressor [39]. The corresponding dynamical system has 16 parameters, 16 functions, and 9 differential equations.
- **Cell Apoptosis Pathway:** The cell apoptosis pathway models the mechanism by which unfolded protein stress response controls the decision mechanism for recovery, adaptation, and apoptosis [14]. The corresponding dynamical system has 87 parameters, 48 functions, and 27 differential equations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Parameters</th>
<th>Functions</th>
<th>Differential Equations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>menten</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>Michaelis-Menten equation reaction [30]</td>
</tr>
<tr>
<td>gentoggle</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>genetic toggle switch in E.coli [18]</td>
</tr>
<tr>
<td>repri</td>
<td>7</td>
<td>3</td>
<td>6</td>
<td>synthetic oscillatory network of transcriptional regulators [11]</td>
</tr>
<tr>
<td>osc</td>
<td>16</td>
<td>16</td>
<td>9</td>
<td>circadian oscillation utilizing activator / repressor [39]</td>
</tr>
<tr>
<td>apop</td>
<td>87</td>
<td>48</td>
<td>27</td>
<td>protein stress response [14]</td>
</tr>
</tbody>
</table>

Table 1. Benchmark Characteristics
<table>
<thead>
<tr>
<th>Component</th>
<th>Quantity</th>
<th>Description</th>
<th>Hardware</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>25</td>
<td>current input</td>
<td>digital to analog converter</td>
<td>Z_D = X_D</td>
</tr>
<tr>
<td>vin</td>
<td>125</td>
<td>voltage input</td>
<td>digital to analog converter</td>
<td>Z_N = X_N</td>
</tr>
<tr>
<td>iout</td>
<td>10</td>
<td>current output</td>
<td>analog to digital converter</td>
<td>Z_O = X_O</td>
</tr>
<tr>
<td>vout</td>
<td>75</td>
<td>voltage output</td>
<td>analog to digital converter</td>
<td>Z_O = X_Y</td>
</tr>
<tr>
<td>vgain</td>
<td>40</td>
<td>voltage gain</td>
<td>summing amplifier</td>
<td>O_Y = (X_Y/Z_Y - 0.04) / Y_V</td>
</tr>
<tr>
<td>iadd</td>
<td>30</td>
<td>current adder</td>
<td>wire join</td>
<td>O_I = A_I + B_I + C_I + D_I</td>
</tr>
<tr>
<td>vadd</td>
<td>35</td>
<td>voltage adder</td>
<td>summing amplifier and capacitor</td>
<td>(\partial O_{2Y}/\partial t = 0.1(A_Y + B_Y - C_Y - D_Y \cdot O_{2Y}))</td>
</tr>
<tr>
<td>vtoi</td>
<td>30</td>
<td>voltage to current converter</td>
<td>operational amplifier</td>
<td>O_I = X_Y/K_Y</td>
</tr>
<tr>
<td>itov</td>
<td>30</td>
<td>current to voltage converter</td>
<td>operational amplifier</td>
<td>O_I = K_Y \cdot X_I</td>
</tr>
<tr>
<td>ihill</td>
<td>8</td>
<td>hill function for activation/repression</td>
<td>logic circuit</td>
<td>S_I = M_Y/(S_I/K_Y)^{p_V} / (S_I/(K_I)^{p_V} + 1)</td>
</tr>
<tr>
<td>igenebind</td>
<td>8</td>
<td>gene binding</td>
<td>logic circuit</td>
<td>O_I = M_I/(1 + K_I \cdot T_I)</td>
</tr>
<tr>
<td>switch</td>
<td>15</td>
<td>genetic switch</td>
<td>logic circuit</td>
<td>O_I = M_I/(S_I/K_I + 1)^{p_V}</td>
</tr>
<tr>
<td>mm</td>
<td>2</td>
<td>Michaelis-Menten dynamics</td>
<td>logic circuit</td>
<td>X_Y = X_0Y - X_Y</td>
</tr>
</tbody>
</table>

Table 2. Utilized Analog Hardware Components and Relations [7, 8, 33, 37, 41].

<table>
<thead>
<tr>
<th>Source Component</th>
<th>Target Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>vgain</td>
<td>outv vadd vtoi mm itov vgain</td>
</tr>
<tr>
<td>iadd</td>
<td>iadd igenebind iout ihill switch genebind itov</td>
</tr>
<tr>
<td>vin</td>
<td>vtoi itov ihill vadd vgxin mm</td>
</tr>
<tr>
<td>vadd</td>
<td>outv vadd vgxin switch</td>
</tr>
<tr>
<td>vtoi</td>
<td>iadd ihill switch iout</td>
</tr>
<tr>
<td>itov</td>
<td>outv vadd vgxin switch</td>
</tr>
<tr>
<td>iiou</td>
<td>iout vtoi vgain vadd</td>
</tr>
<tr>
<td>vtoi</td>
<td>vtoi vadd vgxin switch</td>
</tr>
<tr>
<td>itov</td>
<td>outv vadd vgxin switch</td>
</tr>
<tr>
<td>ihill</td>
<td>iout ihill igenebind switch iadd mm</td>
</tr>
<tr>
<td>igenebind</td>
<td>iout itov igenebind</td>
</tr>
<tr>
<td>switch</td>
<td>iout itov ihill iadd</td>
</tr>
<tr>
<td>mm</td>
<td>outv itov iadd itov</td>
</tr>
</tbody>
</table>

Table 3. Analog Component Connections

5.1 Programmable Analog Device

The specified analog device utilizes mixed voltage and current analog building blocks commonly used for biological network computations [7, 8, 33, 37, 41]. Table 2 describes the input-output relations of the circuit components, the physical analog circuits that implement these relations, and the specified quantity of each kind of component. Table 3 presents the hardware connections that the device supports.

5.2 Quantitative Analysis of Generated Circuits

Table 4 presents, for each benchmark, statistics about the synthesized hardware configuration and the components that the solver uses. Overall, the configurations make good use of the more complex, specialized components. The complexity of the generated circuit generally increases with the complexity of the dynamical system. All of the circuits would be difficult to manually generate.

The number of utilized output components typically matches the number of dynamical system output variables. But because the input components import constants as well as dynamical system input variables into the analog computation, the number of utilized input components is less predictable. The Arco algebraic unification can easily generate new constants, either by (1) evaluating constant subexpressions that arise as the unification maps the dynamical system onto the analog components (we call such constants derived constants), (2) performing unifications that split a single dynamical system constant into multiple constants via algebraic reasoning (we also call these constants derived constants), or (3) materializing new constants as necessary during expression unification to specialize general analog components so that they correctly implement mapped relations from the dynamical system.

Derived constants often adapt dynamical system constants to complex analog component interfaces. Materialized constants often configure general analog components for more specialized use in the synthesized circuit by setting some of their inputs to constant values. All of our benchmarks have both derived and materialized constants. For the larger benchmarks, there are many more derived and materialized constants than constants that appear directly in the dynamical system. We attribute the presence of these derived and materialized constants to (1) the ability of the Arco algebraic unification algorithm to generate these constants as necessary and (2) the fact that many efficient analog building blocks provide complex interfaces that require nontrivial derived and materialized constants for their successful use.

5.3 Materialized and Derived Constants

We next present several examples of materialized (blue) and derived (red) constants in the analog circuits that Arco synthesizes. Black constants are fixed values that are built into the dynamics of the hardware components. These examples highlight how Arco uses algebraic unification to effectively bridge the semantic gap between the dynamical system and the optimized analog hardware components. All of the examples are beyond the reach of standard pattern-based unification systems.
The voltage adder \( v_{\text{add}} \) and the \( \text{vgain} \) component implement the correct multiplication. Here Arco decomposes the parameter 0 into a current adder component to implement \( \text{Perk} \) as:

\[
\frac{1}{(\text{Perk} + 0 - 1 - 0)/1 + 1)}
\]

where 0 and 1 are materialized constants and 4 is a derived constant. Together, these constants specialize the iadd and switch components to perform the proper exponentiation. The iadd component implements the \( \text{Perk} + 0 - 1 - 0 \) subterm. Here Arco synthesizes a multicomponent implementation of a single exponentiation operation.

\section{5.4 Circuit Synthesis Times}

Table 5 presents the running times for synthesizing configurations for the five benchmarks. We performed a linear regression on these times and found that the running time (in minutes) is 0.996573, indicating that, with high confidence, the time varies linearly with the number of relations in the dynamical system + 3.62. The R squared value for the regression is 0.996573, indicating that, with high confidence, the time varies linearly with the number of equations.

Table 5 presents the time required to synthesize configurations for the five benchmarks. We performed a linear regression on these runtimes and found that the running time (in minutes) is 0.996573, indicating that, with high confidence, the time varies linearly with the number of relations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of Equations</th>
<th>Synthesis Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>men</td>
<td>4</td>
<td>5m52.276s</td>
</tr>
<tr>
<td>gg</td>
<td>5</td>
<td>1m11.002s</td>
</tr>
<tr>
<td>re</td>
<td>9</td>
<td>3m34.984s</td>
</tr>
<tr>
<td>os</td>
<td>25</td>
<td>13m13.876s</td>
</tr>
<tr>
<td>ap</td>
<td>75</td>
<td>53m41.902s</td>
</tr>
</tbody>
</table>

\section{6. Related Work}

Much early work in analog computing focused on manually building analog circuits from simple components with the goal of performing numerical computations for simple dynamical systems [10, 27, 32, 38]. This work was based on manual development of the mapping between the problem and the analog circuit. The research presented in this paper, in contrast, focuses on automatic synthesis of analog circuits that use complex components to implement dynamical systems specified in our specification language.
Analog computation has been experiencing a renaissance in the hardware community — modern day programmable analog devices provide complex, domain specific primitives whose behavior is analogous to their physical counterparts [5, 7, 8, 31, 33, 34, 37, 41]. These systems accurately model the dynamical systems they target and minimize discretization errors. The hardware is typically programmed directly at a low level with little or no automation. The techniques presented in this paper, in contrast, automatically synthesize analog circuits from a dynamical system specification.

Recent work in synthesis for analog computing has focused on transistor-level techniques to aid designers in designing specialized analog logic circuits [4, 20, 26]. Our synthesis techniques, in contrast, use the complex multifunctional analog building blocks that these techniques provide to synthesize analog implementations of specified dynamical systems.

Researchers have developed techniques that use analog neural network accelerators to improve the performance of applications written in standard imperative programming languages [16, 36]. These techniques approximately map subcomputations onto trained analog neural network accelerators. Arco, in contrast, maps complete computations onto analog hardware without approximation — the resulting analog configurations are algebraically identical to the specified dynamical system. The goal is not to accelerate an approximable subcomputation expressed in a standard digital programming language — the goal is instead to obtain an algebraically equivalent implementation of the specified dynamical system. To this end, Arco works with a dynamical system specification language (and not a standard digital programming language). The target hardware platform contains specialized components optimized for implementing dynamical systems (and not analog neural networks). There is no training step — Arco synthesizes an exact analog implementation of the dynamical system without training. Any inaccuracy in the solution comes from the inherent analog noise, not from the translation of the computation onto the analog hardware platform. And for our target class of biological dynamical systems, the analog noise is directly analogous to the noise present in the biological system.

The Arco synthesis algorithms use a tableau to organize a search for a configuration of the target analog hardware platform that is algebraically equivalent to the specified dynamical system. The broad concept of a tableau has been widely used in theorem proving [2] and for the synthesis of functional programs [24, 25]. In this context logical deduction rules transform a set of assertions and goals into a proof, potentially with output entries that make it possible to extract a program from the proof. Unlike these previous approaches, Arco works with complex multifunctional analog components, not standard programming language primitives. To correctly utilize these components, Arco uses algebraic unification and must deal successfully with the cascading relation entanglement inherent in the use of such powerful but complex analog components.

Also unlike these previous approaches, the Arco synthesis algorithms operate in the presence of resource constraints — they synthesize the dynamical system onto a hardware platform with finite resources. The Arco synthesis algorithms therefore must track the resources that have been consumed in the synthesis (including complex partially consumed components), with the synthesis failing if it consumes too many resources.

Code generators use tree pattern matching to translate code trees into sequences of machine instructions [3, 12, 17]. Arco, in contrast, works with dynamical systems that may contain feedback loops and circular dependencies. The Denali superoptimizer [21] combines mathematical and machine rewrite rules to generate search spaces of instruction sequences that implement a given computation. It then searches this space to find efficient instruction sequences. Arco, in contrast, produces inherently parallel analog hardware configurations with no concept of sequencing. As noted above, the target components include a finite set of complex, potentially partially utilized analog building blocks optimized for analog efficiency, not digital machine instructions. And the relevant reasoning involves continuous, nonlinear, potentially entangled transcendental functions, not digital logic.

7. Conclusion

Programmable analog devices have emerged as a powerful new computing substrate for scientific computations such as neuromorphic and cytomorphic computations [5, 7, 8, 31, 33, 34, 37, 41]. Arco provides the sophisticated compiler support required to effectively exploit this class of new, powerful, but complex computational platforms.

Acknowledgements

We would like to thank Sung Sik Woo for helpful discussions about programmable analog hardware platforms in general and the specific programmable analog devices that he is building. We would like to thank our shepherd Adrian Sampson and the anonymous reviewers for their very helpful feedback.
A. DSSL Type Checker

A type judgment $\sigma, \gamma \vdash P$ states that the DSSL program $P$ type checks in variable environment $\sigma$ and unit environment $\gamma$.

- **The variable environment** $\sigma$: The variable environment $\sigma : \text{Ident} \rightarrow \text{Type} = \{\text{input}, \text{output}, \text{local}, \text{param}, \text{unit}, \text{time}\}$ tracks the binding of variables to types.

- **The unit environment** $\gamma$: The unit environment $\gamma : \text{Ident} \rightarrow \text{Unit}$ tracks the binding of variables to unit expressions.

Expressions: Figure 10 presents the type checking rules for expressions $E$. A type judgment $\sigma, \gamma \vdash E : U$ states that, with variable environment $\sigma$ and unit environment $\gamma$, $E$ type checks to produce a value measured in units $U$.

The type checking rules ensure only like units are added together and infer new unit expressions for products and quotients. The Add-Type and Sub-Type rules ensure that summands have like units and produce a value measured in the same units. The Multi-Type and Div-Type rules construct a unit expression that is the product or quotient of the unit expressions of the operands of the product or quotient.

Arco supports non-integer exponents only for unitless expressions. The Pow-Type1 rule ensures that if the base expression is not unitless, then the exponent expression must be a unitless integer constant $n$. The Pow-Type2 rule ensures that if the base expression $E_1$ is unitless, then the exponent expression $E_2$ must be a unitless integer.

Unit Expressions: Figure 10 also presents type checking rules for unit expressions $U$. A type judgment $\gamma \vdash U$ states that the unit expression $U$ type checks in unit environment $\gamma$. The Unit-Var, Unit-Mult, Unit-Div, and Unit-None rules check that all of the variables $x$ in the unit expression $U$ are units.

Statements: Figure 11 presents the type checking rules for statements. These rules check that all variable declarations use defined units $U$, there are no variable redeclarations, that relation definitions $\text{rel}$ only define values for output or local variables, and that the units of the defined variables $x$ and the relevant expressions $E$ are consistent. The $\text{eq}(U, U')$ function tests the equivalence of two or more unit expressions $U$ and $U'$. It algebraically simplifies the unit expressions, interpreting none (no unit) as 1, and determines if the simplified expressions are equivalent.
B. Analog Hardware Type Checker

A type judgment \( \sigma, \gamma, \phi \vdash P \) states that the hardware specification \( P \) type checks in variable environment \( \sigma \), unit environment \( \gamma \), and component environment \( \phi \).

- **variable environment** \( \sigma \): The variable environment \( \sigma : \text{Ident} \to \text{Type} \) maps variable identifiers to types, where Type = [time, prop, unit, comp].

- **unit environment** \( \gamma \): The unit environment \( \sigma : \text{Ident} \to \text{Unit} \) maps variable identifiers to unit expressions.

- **component environment** \( \phi \): The component environment \( \sigma = \{ (n, \alpha, \beta) \mid x \in \text{Ident} \} \) maps the component identifier to the component definition information. The parameter environment \( \alpha : \text{Ident} \to \text{Unit} \) maps parameter identifiers to units. The port environment \( \beta : \text{Ident} \to \text{PType} \) maps the port identifier to the port types \( k \in \text{PType} \), where \( \text{PType} = \{ \text{input}, \text{output} \} \). The instance count of the component \( n \in \mathbb{N} \) specifies the number of component instances.

**Port Collection Type Checking Rules**: The \( \text{wf-conn} \) rule in Figure 12 and the rules in Figure 12 check port collections \( C \). A conn \( C \rightarrow C' \) statement specifies that all of the output ports identified by \( C \) can be connected to any of the input ports identified by \( C' \). The hardware specification language supports several kinds of port collections:

- \( x[l:u], x' : \) Port \( x' \) of instances \( l \) to \( u \) of component \( x \).
- \( x[l:u] \): All ports of instances \( l \) to \( u \) of component \( x \).
- \( x, x' : \) All ports of all instances of component \( x \).
- \( *, * : \) All ports of all instances of all components.

The rules check that for a statement conn \( C \rightarrow C' \), \( C \) identifies at least one output port and \( C' \) identifies at least one input port. The rules also check that \( C \) and \( C' \) are well
formed, identified component instances are within the maximum number of instances, and all ports and components are defined.

**Expression Type Checking Rules**: Figure 13 presents the `PORT-TYPE` and `PARAM-TYPE` type checking rules for expressions $E$. The remaining rules include all of the DSSL expression type checking rules from Figure 10 except the `VARIABLE-TYPE` rule. The type judgments are $\sigma, \gamma, \alpha, \beta \vdash E : U$ instead of $\sigma, \gamma \vdash E : U$. A type judgment $\sigma, \gamma, \alpha, \beta \vdash E : U$ states that $E$ type checks to produce a value measured in units $U$.

**Statement Type Checking Rules**: Figure 14 presents the type checking rules for programs $P$. Each rule implements the type checks for a different kind of statement $S$.

**References**


