*Proceedings of the 2023 Winter Simulation Conference*
*C. G. Corlu, S. R. Hunter, H. Lam, B. S. Onggo, J. Shortle, and B. Biller, eds.*

# FACT: A DOMAIN-SPECIFIC LANGUAGE BASED ON A FUNCTIONAL ALGEBRA FOR CONTINUOUS TIME MODELING

Edil Medeiros
Eduardo Peixoto

University of Brasília
North Wing, Electrical Engineering Department
Brasília, DF 70910-900, BRAZIL

Eduardo Lemos

University of Brasília
North Wing, Computer Science Department
Brasília, DF 70910-900, BRAZIL

## ABSTRACT

Hybrid and cyber-physical systems create synergy by combining digital modules with analog implementations of signal processing operations typically implemented in the digital domain. We propose a domain-specific language (DSL), so-called FACT – Functional Algebra for Continuous Time, based on the algebraic properties of the General Purpose Analog Computer (GPAC), a theoretical model of computation recently updated as a continuous time equivalent of the Turing Machine. We lift the GPAC to a *continuous time dynamics inside a black box* semantics for understanding hybrid systems, which allows us to redefine continuous time semantics inspired by the functional reactive programming style. FACT leverages the type class mechanism from the Haskell functional programming language to implement operators that capture the proposed continuous time semantics. An speed-optimized working open-source implementation in the Haskell functional language is provided and was used to demonstrate how the language supports modeling and simulation.

## 1 INTRODUCTION

*"We shape our buildings and afterwards our buildings shape us"* (Churchill 1943). That was how Mr. Churchill addressed the debated question of rebuilding the House of Commons chamber after its destruction during the Blitz. He opposed the semi-circular design favoured by legislative assemblies abroad claiming that the original adversarial rectangular pattern of the chamber was responsible for the two-party system which constitutes the essence of the British parliamentary democracy. He understood the power of legacy to constrain the effectiveness of a decision-making system for many years ahead. Today we feel the troubles of legacy in the area of hybrid and cyber-physical systems (CPS) despite our apparent success to deliver automation systems for industrial, medical, avionics, automotive, and other application domains.

Engineering strives to build artifacts that faithfully mimic a chosen *model* which serves as the *specification* for how the artifact should behave (Lee 2016). Entrenched in its practices are time-proven modeling frameworks that provide reasoning tools, i.e., *abstractions* derived from different areas of mathematics, each of which developed and employed with different sets of problems in mind. Of particular interest for CPS design is the challenge of combining the *continuous-time* (CT) *dynamics*, used to understand the physical world by means of ordinary differential equations (ODE), with the *discrete-time* (DT) *nature of computations* performed by digital computers and the untimed models used to specify their behavior as software. The clash of *incompatible notions of time* makes the interaction between them hard to formalize in a deterministic useful way (Lee 2016), although effort has been addressed to some degree by Lee (2014), Ungureanu et al. (2018), Attarzadeh-Niaki and Sander (2020), Ungureanu et al. (2021).

Because digital computers have been so successful, they serve as buildings that shape our understanding of computers as machines and algorithms as processes; it is now difficult to understand what CT processing

is about. In the willingness to get technical advantages of speed and precision, we traded realism — the close correspondence between the thing being modeled and the computer model (Nyce 1996). Yet, we see the resurgence of the idea of continuous time as a proper model of computation to expand and unify classical discrete time concepts like computability and computational complexity (Silva Graça 2004; Bournez et al. 2017) as well as in the implementation of actual analog computers intended to serve as co-processors for specialized computations. Such hybrid systems, i.e., systems composed of both analog and digital parts, explore the no-time discretization property present in analog circuits while being assisted by the calibration provided by its digital counterpart. This strategy yields several benefits, such as faster solutions, a better interface with measure instruments, and the absence of convergence issues when dealing with continuous systems. Hybrid-based solutions can be now found in numerous fields, such as robotics (Guo et al. 2015), modeling nonlinear systems (Cowan et al. 2005) and neuromorphic computing (Cramer et al. 2022).

In this paper we present FACT (Functional Algebra for Continuous Time Modeling), an open-source (Lemos and de Medeiros 2022) domain-specific language (DSL) based on the continuous time formalism presented by de Medeiros et al. (2018). In that work, the authors attack the continuous-time modeling problem by defining a set of basic functional units and a set of composition rules with well defined semantics, regarded as axioms and operations in a formal algebraic system. The chosen rules abstract explicit signal manipulation in the final modeling language which, despite its mathematical elegance, pose practical difficulties on expressiveness and designer productivity we aim to solve in the current proposal. Inspired by the Aivika multi-method simulation library (Sorokin 2021), FACT goes in the opposite direction by exposing time varying signals which we show to enable a direct translation of sets of differential equations into an executable model. The discipline required by the underlying formal system is imposed by leveraging Haskell's strong type system as guidance.

## 2 THE GENERAL PURPOSE ANALOG COMPUTER

The General Purpose Analog Computer (GPAC) (Shannon 1941) is a *model of computation* (MoC) originally proposed in 1941 by Claude Shannon to formalize the operation of the Differential Analyzer (Bush 1931), a popular mechanical device of the 1930s intended to solve numerical problems. Shannon shows how the intricate interaction between gears and shafts on the machine culminates in solving a particular class of differential algebraic equations within the continuous time domain, finding fundamental limits of the model and thus of its physical implementation.

The GPAC is a mathematical model sustained by proofs and axioms about a set of basic units (or circuits), shown in Figure 1, and their composition rules (de Medeiros et al. 2018). The *constant unit* generates $f(t) = k$, a real constant output for any time $t$; the *adder unit* generates $f(t) = u(t) + v(t)$, the sum of two given inputs with both varying in time; the *multiplier unit* generates $f(t) = u(t)v(t)$, the product of two given inputs varying in time; lastly, given an input $u(t)$ and an initial condition $w_0 = u(t_0)$ at time $t_0$, the *integrator unit* generates the output $w(t) = w_0 + \int_{t_0}^{t} u(t)\,dt$, where $u$ is called the *integrand*.

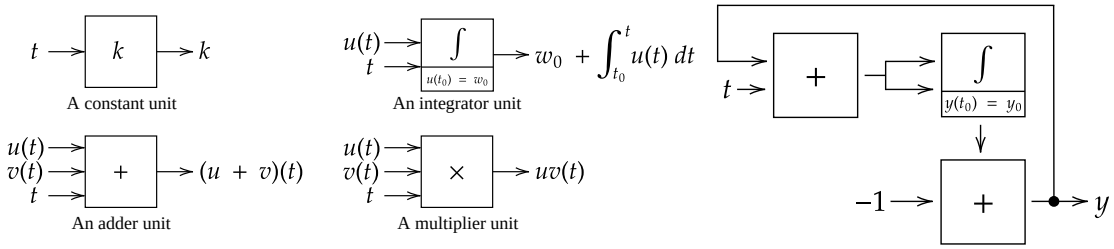



Figure 1: GPAC basic units.




Figure 2: Nondeterministic machine.

Composition rules restrict how these units can be hooked to one another. Shannon (1941) established that a valid GPAC is one that: 1) for each unit, two inputs and two outputs are not interconnected (short-circuited), and 2) the inputs are only driven by either the independent variable $t$ (regarded as the *time*) or by a single unit output. The former rule is intended to forbid nondeterministic systems with multiple solutions that would lack a physical implementation, like the one shown in Figure 2 which yields as solution $y(t) = 1 \pm \sqrt{-2t} - t$ (Silva Graça and Félix Costa 2003). The GPAC extension proposed by Silva Graça (2004), the FF-GPAC, added new constraints defining no-feedback (acyclic) GPAC configurations to be *polynomial circuits* by using only constant function units, adders and multipliers. Thus, FF-GPAC's composition rules, which we assume to be the canonical set, are:

- Each polynomial circuit admit multiple inputs;
- An input of a polynomial circuit should be the input $t$ or the output of an integrator;
- Each integrand input of an integrator should be generated by the output of a polynomial unit;

The polynomial circuits $A_k$, as shown in Figure 3, compute values *point-wise* in respect to their inputs, thus resembling *combinational* circuits. In contrast, integrators impose a dynamic behavior for the system because their outputs depend on previously computed values, implying memory-dependent behavior or *sequential* circuits. Feedback can only be achieved from the output of integrators to inputs of polynomial circuits. Hence, pathological algebraic systems like $x = x + 1$ which does not have a deterministic solution are avoided. The final GPAC topology will resemble a register transfer level model, as shown in Figure 4, in which time is the only input from the top-level perspective.

All algebraic functions (e.g. quotients of polynomials and irrational algebraic functions) and algebraic-transcendental functions (e.g. exponentials, logarithms, trigonometric, Bessel, elliptic and probability functions) can be described using the FF-GPAC model. Additionally, the class of functions that are generable by the GPAC is closed under the usual arithmetic operations and thus one can use any such generated function just like any basic unit, e.g., $\dot{w}(t) = 5\sin(w(t))$ is generable because $\sin(t)$ is in the class of generable functions (Bournez et al. 2016). This property of the FF-GPAC has an essential implication for engineering applications: it enables hierarchical modeling. The composition rules imply that we can only define a single feedback loop, although it can be a complex feedback involving all state variables at once. This, in fact, establishes one level of a possibly multilevel GPAC circuit in which each level acts as a black box. This black box can be used in the definition of another GPAC, with feedback in a higher hierarchical level, like the one shown in Figure 5.

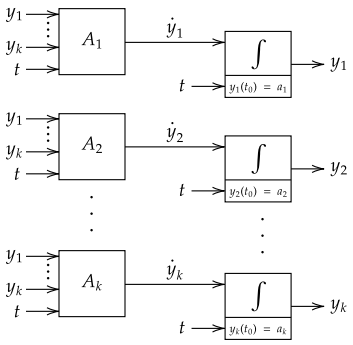



Figure 3: GPAC topology.

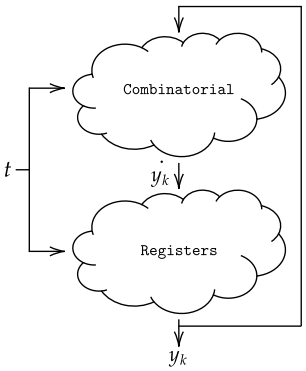



Figure 4: GPAC as an equivalent of register transfer level models.

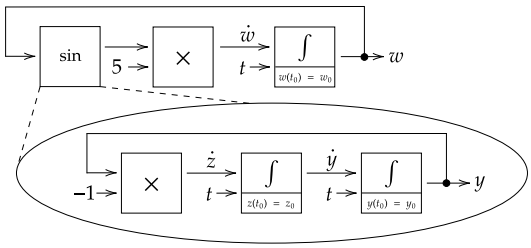



Figure 5: Hierarchical GPAC machine.

### 2.1 GPAC as a Functional DSL

In 1966, Landin (1966) identified there were already 1700 specialized programming languages used to express concepts in over 700 application areas. These abundance reflected the tension between conflicting requirements for system design languages: (1) minimality; (2) mathematical rigor; and (3) executability (Sifakis 2013). In the context of CPS, we find the need for expressing multifaceted behaviors on one hand, and the potential complexity and scalability issues of capturing too many aspects of CPS design within one single language on the other. When evaluating his contemporaries, however, Landin proposes that we design *vocabularies*, instead of full-fledged languages, and host them within consistent, well defined frameworks thus giving rise to the concept of *embedded domain-specific languages* (EDSL).

Following Landin, Backus (1978) in his Turing Award lecture argues that a functional paradigm powered by an *algebra of combining forms* liberates designers from the *von Neumann style* of thinking about systems as transitions between states which, inherited as a modeling paradigm, contributes to degenerate the description of aspects such as continuum that transcend the underlying tool used to analyze an engineering model. The ForSyDe-Atom framework (Ungureanu et al. 2021) is a framework for modeling CPS that encapsulate multiple MoC semantics such as synchronous (SY), discrete event (DE), continuous time (CT), and synchronous dataflow (SDF) (Jantsch 2003). It proposes a functional paradigm based on three mechanisms that discipline and unifies the use of different DSLs for a coherent design flow:

- *layers* that syntactically separate different aspects of CPS in interacting DSLs;
- *atoms* acting as primitive, composable building blocks to define these DSLs;
- and *patterns* that handle complexity in a compositional, hierarchical manner.

We understand the GPAC to be a preferable formalization for the CT MoC presented in (Ungureanu et al. 2021) which lacks an underlying formal mathematical foundation. In what follows, we provide an implementation and semantics for a novel DSL that allows modeling of CT systems. It provides the mechanisms required by the ForSyDe-Atom framework to define a layer, viz., a structured data type $CT_\alpha$; a finite set of atoms with clearly defined semantics; among these, one ($\text{map}_{CT}$) that lifts pure functions into instances in the DSL; and a set of meaningful composition rules (the GPAC composition rules) enforced by the library. We focus the present work on detailing the FACT language and its use in isolation for CT modeling and regard hybrid simulations to a future work.

## 3 CONTINUOUS TIME SEMANTICS

To formalize a compatible interface between the continuous time domain and its discrete counterpart we adopted the following semantics. *A system's continuous-time dynamics happen inside a black box that can be asked for the state of the system at an arbitrary time* $t \in \mathbb{R}$. Think of it as a mechanism hidden inside a safe box. When the box is open, an operator may configure the system's topology and initial conditions, but it can't turn the machine on. On the other hand, when the box is closed, the device evolves its state over time, but the operator has no access to any information about what's happening inside. He can only decide to open the box, putting it to a halt, and inspect the final state of the system. In this model, the continuous time system *reacts* to a discrete operator whose function is to drive the machine state evaluations. The operator logic could be described by a discrete time formalism such as classical state machines like in (Lee and Zheng 2005). In what follows we describe static continuous time machines, those situations in which the operator chooses to never change the machine configuration once it is put on the black box.

It remains to define the continuous time operational mechanics, i.e., which process will *implement* the aforementioned definition. We choose to rely on numerical methods, hiding from a system designer the implementation details behind functional data structures that represent the continuous time semantics. Inspired by the concept of tagged systems (Derler et al. 2012), we define a *continuous time machine* (CT) as a *function*

$$CT_\alpha : \rho \to \alpha \tag{1}$$

from an embellished time tag $\rho$ to an arbitrary output value $\alpha$. The notation : describes the type of such function. We make the CT machine polymorphic with respect to its output type regarding to the designer the decision of how to best represent an output in the domain of interest. FACT takes care only of the time behavior of the computations while allowing to lift functional abstractions into the DSL as described in the next section.

The embellished tag carries the relevant information required for simulation of the continuous time machine, lifting an *ordinary differential equation solver* to define an operational semantics for the model. Fig. 6 shows the structure of the tag as a 4-tuple. The fields `Time` and `Interval`, both carrying values within the real numbers $\mathbb{R}$ domain, represent the current moment of the simulation being computed and the time interval of interest in which the system will be analyzed. The `Solver` field carries the information required to drive the specific ODE solver that will be employed in the simulation of the system. A numerical ODE solver requires a *time step* `dt` in $\mathbb{R}$ used to advance in time to $t + dt$. The set $\mathbb{M}$ describes the collection of algorithms available to the designer. Currently our implementation supports the single-step Euler method, the second and forth order multi-stage Runge-Kutta methods. Other explicit or implicit methods as well as adaptive time step algorithms can be added to the library to extend its capabilities. The set $\mathbb{S}$ holds the set of natural numbers that describe all possible stages' indices used by the available solver methods and the `Iteration` field keep track of the solver work on a iteration domain. This approach of making the solver part of the model context specification allows different parts of a system to be described and simulated using different algorithms while ensuring some level of semantic isolation between modules. We regard a deeper analysis of this possibility as a future work.

We define an interpretation of CT machines in the context of a solver as a function from *time* to values

$$\text{at}_{\text{Solver}} : CT_\alpha \to \tau \to \alpha$$

consulting the dynamics of the CT machine at an arbitrary time $\tau$ and producing its output of type $\alpha$. Under the hood, based on the time step and solver information provided, FACT will imply a discrete time axis where the numerical ODE solver will perform. Refer to the `Iterations` axis in Fig. 7. A simulation is then interpolated and projected back into the continuous time domain that is exposed to the user. The final effect is that of a mechanism running in a black box as per our chosen metaphor. Thus $\text{at}_{\text{solver}}$ drives a simulation and other drivers that expose the intermediate simulation steps are derived from it.

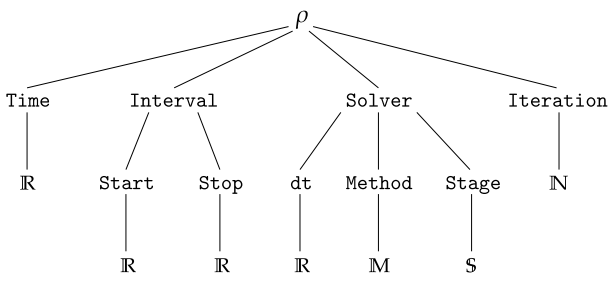



Figure 6: The $\rho$ tag carries multiple types of information.

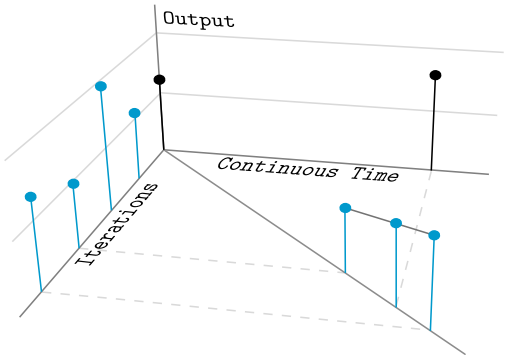



Figure 7: A simulation is performed in the `Iterations` axis and projected back into the continuous time domain.

## 4 UPLIFTING A PROGRAMMING INTERFACE

We define a *structured data type* as a type built using a *type constructor*, i.e., a function of the form $\alpha \to CT_\alpha$ that wraps an arbitrary value into a CT machine thus enabling its model behavior when evaluated.

To construct CT machines we define $const_{CT}$ that builds a map from a tag $p : \rho$ to a constant value $k : \alpha$ and captures the semantics of GPAC's constant unit. The notation $\mapsto$ builds a function, so-called *anonymous function*, that, in the case of $const_{CT}$, expects a value $p$ and return $k$, i.e., a CT machine containing a value with the same type $\alpha$ established by $k$.

$$\begin{aligned} &\text{const}_{CT} : \alpha \to CT_\alpha \\ &\text{const}_{CT}\ k = p \mapsto k \end{aligned} \tag{2}$$

To implement the adder and multiplier units we employ the notion of *higher-order functions* (HOF) that lifts functions into the domain of the CT machines. The first is $map_{CT}$ that evaluates a CT machine $m_1$ with the tag $p$ and transforms its output with a function $f$.

$$\begin{aligned} &\text{map}_{CT} : (\alpha \to \beta) \to CT_\alpha \to CT_\beta \\ &\text{map}_{CT}\ f\ m_1 = p \mapsto f(m_1(p)) \end{aligned}$$

Next is $merge_{CT}$ in which the function $f$ is yielded by another CT machine $m_f$. Figs. 8 and 9 depict these operations schematically. Note that in both cases the evaluation of the internal CT machines use the same tag $p$ of the resulting machine meaning they are evaluated synchronously.

$$\begin{aligned} &\text{merge}_{CT} : CT_{(\alpha\to\beta)} \to CT_\alpha \to CT_\beta \\ &\text{merge}_{CT}\ m_f\ m_1 = p \mapsto f(m_1(p)) \\ &\quad \text{where} \quad f = m_f(p) \end{aligned}$$

For brevity, we introduce the infix notation

$$\begin{aligned} f \odot m_1 &= \text{map}_{CT}\ f\ m_1 \\ m_f \otimes m_1 &= \text{merge}_{CT}\ m_f\ m_1. \end{aligned}$$

In this way, we define GPAC's adder and multiplier units as

$$\begin{aligned} &\text{add}_{CT} : CT_\alpha \to CT_\alpha \to CT_\alpha \\ &\text{add}_{CT}\ m_1\ m_2 = (+) \odot m_1 \otimes m_2 \end{aligned}$$

and

$$\begin{aligned} &\text{mult}_{CT} : CT_\alpha \to CT_\alpha \to CT_\alpha \\ &\text{mult}_{CT}\ m_1\ m_2 = (\times) \odot m_1 \otimes m_2 \end{aligned} \tag{3}$$

for all CT machines that produce outputs in which addition and multiplication are well defined, e.g., integers, floating-point numbers, square numerical matrices.

In the actual implementation of FACT, these are the functions required to implement Haskell's `Functor` and `Applicative` *type classes* used to implement other host language's primitives. A type class can be thought of as an abstract data type (Wadler and Blott 1989), i.e., a set of valid data equipped with standard functions that make sense on the type's context. Haskell type classes permit overloading of language operators enabling *ad-hoc polymorphism*, i.e., allowing a function to be defined over several types, acting in a different fashion for each one. In our case we define the required type classes such that the CT machines that yield numerical outputs inherit the host language syntax for numerical operations. Thus

```
m1 = (1 + 2) * 3
```

will be automatically translated by the host language into the CT machine shown in Figure 11.

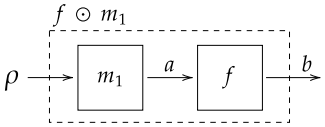



Figure 8: $\text{map}_{CT}$ operation.

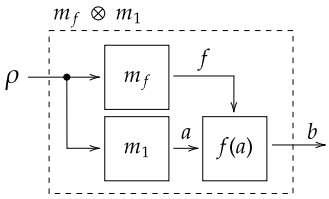



Figure 9: $\text{merge}_{CT}$ operation.

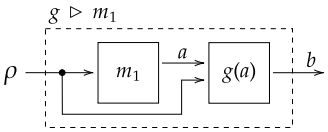



Figure 10: $\text{bind}_{CT}$ operation.

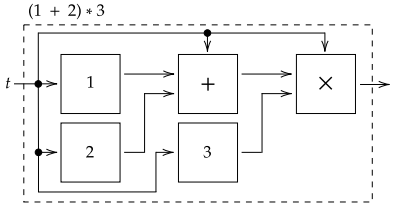



Figure 11: Combinational GPAC circuit.

### 4.1 Sequential units

The GPAC's integrator is the unit that introduces a notion of memory into a CT machine requiring an initial value and an integrand to be fully defined. Hence, we define the integrator unit as the tuple

$$\mathrm{I}_\alpha : CT_\alpha \times CT_\alpha \times CT_\alpha.$$

Because it is composed out of CT machines, time is always available as an input to an integrator in terms of its GPAC representation. Its first term carries a constant unit to represent an initial value $w_0$, the second term, so-called `computation`, carries a CT machine to represent an integrand $u(t)$, and the third term, named `cache`, is used for a memoization mechanism. Internally, the latter two terms are pointers that establish an *implicit recursion* between each other, i.e., the pointer `computation` needs to have access to previously computed values due the solvers' nature of always using past outputs to compute the value of the current iteration. Without this optimization, every single step of each solver method would have to re-calculate all the previous steps until it reaches the initial condition of the system. This would impact performance, both in time and memory usage, due to all the recursive calls required at each step. Hence, these values are stored in a memory location that the pointer `cache` grants access to. Furthermore, these past values requires knowledge of the integrand, i.e., the pointer `cache` computes these values by reading the integrand present in the `computation` pointer. This mechanism is encapsulated by the DSL, making this pointer manipulation transparent from an usability perspective.

Inspired by the CRUD (Create, Read, Update, Delete) strategy, common in relational databases, the following functions describe how interactions with an integrator can be managed within the DSL:

$$\begin{aligned}
&\text{createInteg}_{CT} : CT_\alpha \to CT_{I_\alpha} \\
&\text{readInteg}_{CT} : I_\alpha \to CT_\alpha \\
&\text{updateInteg}_{CT} : I_\alpha \to CT_\alpha \to CT_{I_\alpha}
\end{aligned}$$

The function $\text{createInteg}_{CT}$ takes a constant unit as the initial condition and introduces an integrator in the context of a CT machine. Also, it initializes the two aforementioned pointers, e.g., the pointer `cache` is set to read from the pointer `computation`, whilst prepares the memoization table in memory. The function $\text{readInteg}_{CT}$ enables one to acquire the current state of the integrator via reading the output value stored in the `cache` pointer. In FACT, this is used to define the system's *state variables* as we show in the next section. Finally, the function $\text{updateInteg}_{CT}$ takes an integrator and an integrand described as a CT machine and updates the integrator structure, i.e., it updates the value stored in the `computation` pointer by identifying the appropriate ODE solver. When picking a solver, previous values are required

and are accessed via reading the `cache` pointer. The deletion operation is handled by Haskell's automatic memory management runtime (garbage collection).

Note that the integrator manipulation functions' types generate structure as the outcome, i.e., they all wrap their results in a CT machine, although their arguments are pure values. The required lifting of the integrator functions is performed by an additional HOF $\text{bind}_{CT}$

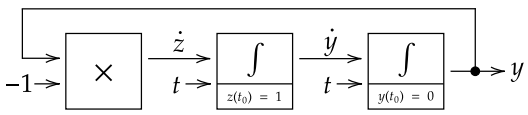
$$\begin{aligned} &\text{bind}_{CT} : (\alpha \to CT_\beta) \to CT_\alpha \to CT_\beta \\ &\text{bind}_{CT}\ g\ m_1 = p \mapsto m_2(p) \\ &\quad \text{where} \quad m_2 = g(m_1(p)) \end{aligned}$$

that evaluates a machine $m_1$ to get a useful value for an input function $g$ able to produce a new CT machine. Figure 10 shows an schematic interpretation of $\text{bind}_{CT}$ in which the execution of the internal $m_1$ CT machine is required to drive the execution of a second internal CT machine produced by $g$. In practice $\text{bind}_{CT}$ formalizes the notion of sequencing operations in the context of CT machines. We introduce the infix notation $f \triangleright m_1 = \text{bind}_{CT}\ f\ m_1$ for convenience. Additionally, $\text{bind}_{CT}$ is required to define the `Monad` type class in Haskell, unlocking additional syntactic support on the host language to manipulate CT machines.

## 5 FACTORIZATION OF ODES

Every GPAC composition is equivalent to a system of ordinary differential equations, each being represented by a CT machine in FACT. Thus, a system of equations, so-called a *model*, is composed by multiple machines *wired* together. The set of tangled smaller machines creates a CT machine, detailed by Equation 1, that describes the entire system. Figure 12 shows a GPAC machine that computes sine and cosine functions (Bournez, Graça, and Pouly 2017) as an example of the look and feel of a model in FACT. The accompanying code is the description of this system in FACT.

```
1  sineGPAC = do
2    integY <- createInteg 0
3    integZ <- createInteg 1
4    let y = readInteg integY
5        z = readInteg integZ
6    updateInteg integY z
7    updateInteg integZ (-1 * y)
8    return $ sequence [y, z]
```




$$\begin{cases} \dot{y}(t) = z(t) \\ \dot{z}(t) = -y(t) \end{cases}$$

Figure 12: GPAC configuration that represents the ODE system that culminates in $y(t) = \sin(t)$ with its FACT implementation.

Haskell desugars `do` blocks using the defined $bind_{CT}$ to perform the model operations in the sequence their appear in the source code. Lines 2-3 introduce the two integrators into the model with their corresponding initial conditions. Each integrator also initialized their internal pointers. Lines 4-5 define names to refer to the outputs of the integrators, i.e., the state variables of the continuous time system that read from the their respective `cache` pointer. Those are the functions $y(t)$ and $z(t)$ we are interested in evaluating. Lines 6-7 install the differential equations that describe the system dynamics into the model. This installation process updates the values of the internal points, which establishes a recursive mechanism internally. The `-1` constant is desugared by the host language into a GPAC constant unit as in Equation 2 and the multiplication operator `*` into $mult_{CT}$ from Equation 3. Finally, line 8 uses the Haskell `sequence` function to collect the results in a list that is returned as the result of the model evaluation. Such model being a CT machine, implies that a tag needs to be generated and applied to the model in order to gather the generated outputs.

The DSL exports *driver* functions that fulfill this role. The functions $runCTFinal_\alpha$ and $runCT_\alpha$ generate appropriate tags according to the simulation setup and apply it into the provided model. The former function

runs the simulation until it reaches the time of interest, whilst the latter function outputs all the intermediate values between the start of the simulation and final time of interest, with the values interspaced by the configured time step. These driver functions consider the interpolation logic detailed in section 3 (Figure7) when creating the tag that will trigger the simulation. Below, there are the type signatures of these two aforementioned driver functions:

$$\text{runCTFinal}_{\alpha} : \mathit{Model}\ \alpha \to \alpha \to \mathit{Solver} \to \alpha$$
$$\text{runCT}_{\alpha} : \mathit{Model}\ \alpha \to \alpha \to \mathit{Solver} \to [\alpha].$$

These functions expect to receive a model, such as the `sineGPAC` model, a specific point in time that the simulation will stop and which has been parameterized with the$\alpha$ type, and the solver's configuration, which contains the solver method and size of the time step.

We chose to present the same example as in (de Medeiros, Ungureanu, and Sander 2018) to allow one to compare both DSLs. In that work the authors abstract the wiring between the basic units by providing a set of operators to the designer and the implementation of this same example resembles the direct translation of the block diagram into a textual form. FACT, on the other hand, abstracts the manipulation of the wiring diagram by allowing the designer to express a model by a translating a familiar set of differential equations.

All models described by a system of differential equations in FACT will follow the same pattern detailed in Figure 13. The integrator operations expose operations on the semantics domain visible to the designer. Under the hood, it performs low-level operations on the operational domain on the language. One defines the system integrators and initial conditions using createInteg$_{CT}$ and FACT allocates memory on the host machine. State variables are defined by readInteg$_{CT}$ with FACT providing pointers to them. The differential equations are installed by updateInteg$_{CT}$. Finally, collecting the state variables drives the model execution. This could be further automated by a compiler that reads differential equations descriptions and uses FACT as an intermediate language to create an executable model.

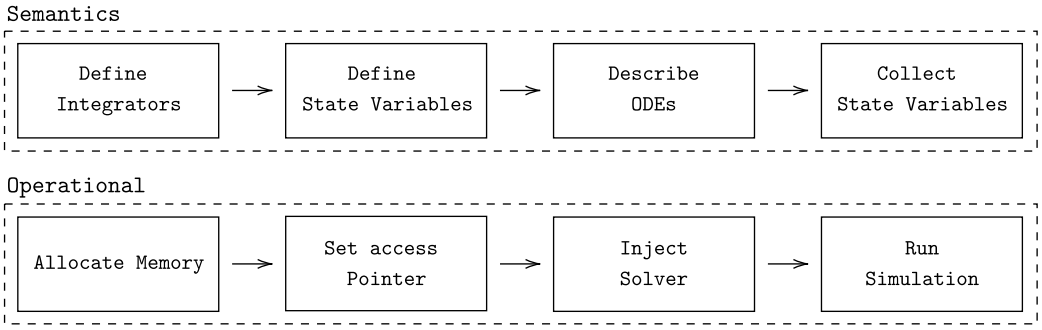



Figure 13: FACT exposed semantics and operational domain.

As a slightly more involved example, the classic Lorenz's Attractor system is represented as the GPAC diagram shown in Figure 14, in which the time input was omitted for the sake of removing visual pollution. This is an example of a coupled feedback loop system in which each state variable potentially depends on every other state variables. In this case state variables $x(t)$ and $y(t)$ have feedback paths back to each one of the three integrators in the system and the state variable $z(t)$ to two integrators. Such a topology presented a challenge to model with the DSL proposed by de Medeiros et al. (2018) that only provided an operator for single-path feedback loops representing decoupled systems. This is no problem in our DSL as shown in the following model. Not every system can be decoupled by semantic preserving transformations of GPAC diagrams, suggesting that the proposed set of primitives of FACT are better suited to fully capture GPAC's expressiveness power.

Finally, we made a test using the Lorenz attractor model for assessing a more complex system using the DSL. Figure 15 shows the output of the model. The test bench simulated the system of ODEs from 0

to 100 seconds, using the second order Runge-Kutta method. Figure 16 presents the execution times of the model running in a Ryzen 7 5700X CPU when decreasing the size of the time step, which increases the total number of iterations. The graph suggests that our current implementation yields a linear complexity executable model in regard to the number of iterations. We plan to investigate more intricate scenarios in the future in which the number of integrators in the model increase, allowing us to analyze complexity in that dimension.

```
lorenzGPAC = do
 integX <- createInteg 1.0
 integY <- createInteg 1.0
 integZ <- createInteg 1.0
 let x = readInteg integX
    y = readInteg integY
    z = readInteg integZ
    sigma = 10.0
    rho = 28.0
    beta = 8.0 / 3.0
 updateInteg integX (sigma * (y - x))
 updateInteg integY (x * (rho - z) - y)
 updateInteg integZ (x * y - beta * z)
 return $ sequence [x, y, z]
```

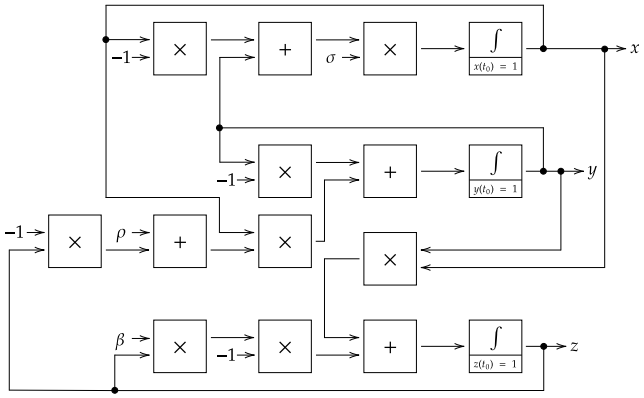



Figure 14: GPAC configuration of the classical Lorenz Attractor with its FACT implementation.

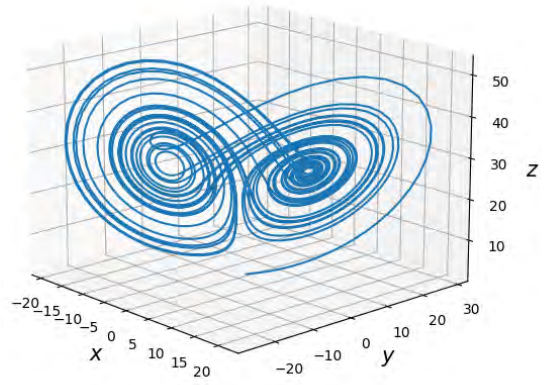



Figure 15: Plot of the Lorenz attractor model.

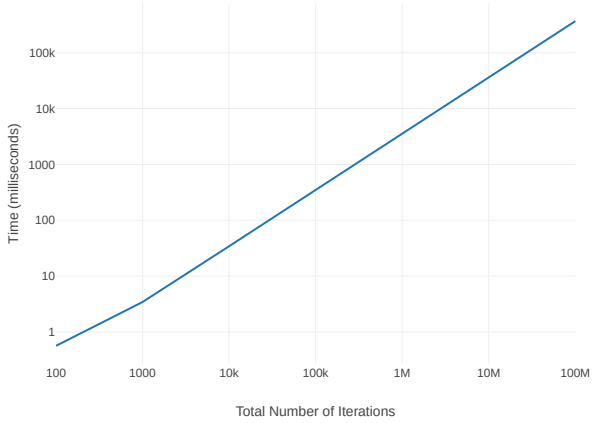



Figure 16: Performance metrics of FACT.

## 6 RELATED WORK

Similar strategies using formal means used in FACT were previously proposed by the EDA research community. CyPhySim (Lee, Niknami, Nouidui, and Wetter 2015) concerns multi-domain simulation by means of a hierarchical notion of time, including continuous-time dynamics. While it is based on the notion of a central director to ensure conformance of semantics, CT machines in FACT encapsulate continuous-time semantics and operate on a local notion of time. Its notion of modal models where CT systems are mixed within classical finite state machines are the inspiration for the extensions we plan for FACT to interact with discrete time systems.

ForSyDe-Atom (Ungureanu et al. 2021) is a framework inspired by functional programming ideas and while it includes a layer dedicated to continuous-time modeling, it does not detail how dynamic

systems are represented, taking for granted the dynamics itself and defining an interface to interact with other formalisms. FACT primitives are similar to the Atoms proposed in the work, suggesting it could be integrated to extended the capabilities of that tool.

Zélus (Bourke and Pouzet 2013) extends the Lustre language with formal constructs to ensure discrete computations are aligned with specific events happening on the continuous domain. In practice, it implements a collision detection algorithm that generates events the discrete domain react to. In FACT we propose, instead, that CT machines react to discrete time events to model more closely practical hybrid systems where digital subsystems define how and when to monitor and act on the continuous time parts.

Finally, FACT take many ideas from the functional reactive programming paradigm like in Chupin and Nilsson (2019), Perez and Nilsson (2020). The main similarity is to represent continuum as symbolic abstractions where numerical representations become apparent only in a late stage of model evaluation.

## 7 CONCLUSION AND FUTURE WORKS

We presented FACT — Functional Algebra for Continuous Time Modeling — a domain-specific language based on the continuous time formalism of GPAC. FACT defines an operational semantic for continuous time systems and hides it from users, exposing functional primitives that allow designers to think in terms of familiar systems of differential equations instead of dealing with the digital tools and algorithms used to evaluate the models. An speed-optimized working open-source implementation in the Haskell functional language is provided and was used to demonstrate how the language supports modeling and simulation on early stages of a design flow. A future work is to extend (or interface) FACT with discrete time formal modeling primitives to unleash the modeling and simulation of all parts of a hybrid system under a unifying formalism.

One of our main concerns so far was the correctness of FACT between its specification and its implementation. Shannon's GPAC concept acted as the specification for the language, whilst the proposed software attempted to implement it. The criteria used to verify that the software fulfilled its goal were by using it for simulation and via code inspection, both of which are based on human analysis. As future work we plan to perform the formal verification of our implementation against GPAC properties to ensure the language does not introduce unintended transformations when modeling a given system of differential equations hence establishing a solid map between specification and its implementation.

## REFERENCES

Attarzadeh-Niaki, S.-H., and I. Sander. 2020. "Heterogeneous Co-Simulation for Embedded and Cyber-Physical Systems Design". *SIMULATION* 96(9):753–765.

Backus, J. 1978. "Can Programming Be Liberated From the Von Neumann Style?: a Functional Style and Its Algebra of Programs". *Communications of the ACM* 21(8):613–641.

Bourke, T., and M. Pouzet. 2013. "Zélus: A Synchronous Language with ODEs". In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, HSCC '13, 113–118. New York, NY, USA: Association for Computing Machinery.

Bournez, O., D. S. Graça, and A. Pouly. 2017. "Polynomial Time Corresponds to Solutions of Polynomial Ordinary Differential Equations of Polynomial Length". *Journal of the ACM* 64(6):1–76.

Bournez, O., D. Graça, and A. Pouly. 2016. "On the Functions Generated by the General Purpose Analog Computer". *Information and Computation* 257.

Bush, V. 1931. "The Differential Analyzer. A New Machine For Solving Differential Equations". *Journal of the Franklin Institute* 212(4):447–488.

Chupin, G., and H. Nilsson. 2019. "Functional Reactive Programming, Restated". In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, PPDP '19, 1–14. New York, NY, USA: Association for Computing Machinery.

Churchill, W. 1943. *HC Deb 28 October 1943 (House Of Commons Rebuilding)*, Volume 393 of *5th*. London: Bantam.

Cowan, G., R. Melville, and Y. Tsividis. 2005. "A VLSI analog computer/math co-processor for a digital computer". In *International Solid-State Circuits Conference. IEEE International Digest of Technical Papers*, Volume 1, 82–586.

Cramer, B., S. Billaudelle, S. Kanya, A. Leibfried, A. Grübl, V. Karasenko, C. Pehle, K. Schreiber, Y. Stradmann, J. Weis, J. Schemmel, and F. Zenke. 2022. "Surrogate Gradients For Analog Neuromorphic Computing". *Proceedings of the National Academy of Sciences* 119(4):1–9.

de Medeiros, J. E. G., G. Ungureanu, and I. Sander. 2018. "An Algebra For Modeling Continuous Time Systems". In *2018 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 861–864.

Derler, P., E. A. Lee, and A. Sangiovanni Vincentelli. 2012. "Modeling Cyber-Physical Systems". *Proceedings of the IEEE* 100(1):13–28.

Guo, N., Y. Huang, T. Mai, S. Patil, C. Cao, M. Seok, S. Sethumadhavan, and Y. Tsividis. 2015. "Continuous-time Hybrid Computation With Programmable Nonlinearities". In *41st European Solid-State Device Research Conference (European Solid-State Circuits Conference)*, 279–282.

Jantsch, A. 2003. *Modeling Embedded Systems And Soc's: Concurrency And Time In Models Of Computation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Landin, P. J. 1966. "The Next 700 Programming Languages". *Communications of the ACM* 9(3):157–166.

Lee, E. A. 2014. "Constructive Models of Discrete and Continuous Physical Phenomena". *IEEE Access* 2:797–821.

Lee, E. A. 2016. "Fundamental Limits Of Cyber-Physical Systems Modeling". *ACM Transactions on Cyber-Physical Systems* 1(1).

Lee, E. A., M. Niknami, T. S. Nouidui, and M. Wetter. 2015. "Modeling And Simulating Cyber-Physical Systems Using CyPhySim". In *2015 International Conference on Embedded Software (EMSOFT)*, 115–124.

Lee, E. A., and H. Zheng. 2005. "Operational Semantics Of Hybrid Systems.". In *Hybrid Systems: Computation and Control*, Volume 5, 25–53. Berlin, Heidelberg: Springer Berlin Heidelberg.

Lemos, E. and J. E. G. de Medeiros 2022. "FACT: Functional Algebra For Continuous Time". https://github.com/FP-Modeling/fact/releases/tag/2.0.

Nyce, J. 1996. "Guest Editor's Introduction". *IEEE Annals of the History of Computing* 18(4):1–3.

Perez, I., and H. Nilsson. 2020. "Runtime Verification And Validation Of Functional Reactive Systems". *Journal of Functional Programming* 30:1–28.

Shannon, C. E. 1941. "Mathematical Theory Of The Differential Analyzer". *Journal of Mathematics and Physics* 20(1–4):337–354.

Sifakis, J. 2013. "Rigorous System Design". *Foundations and Trends in Electronic Design Automation* 6(4):293–362.

Silva Graça, D. 2004. "Some Recent Developments On Shannon's General Purpose Analog Computer". *Mathematical Logic Quarterly* 50(4-5):473–485.

Silva Graça, D., and J. Félix Costa. 2003. "Analog Computers And Recursive Functions Over The Reals". *Journal of Complexity* 19(5):644–664.

Sorokin, D. 2021. "Aivika: A Multi-Method Simulation Library". https://hackage.haskell.org/package/aivika.

Ungureanu, G., J. E. G. de Medeiros, and I. Sander. 2018. "Bridging Discrete And Continuous Time Models With Atoms". In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 277–280.

Ungureanu, G., J. E. G. de Medeiros, T. Sundström, I. Söderquist, A. Åhlander, and I. Sander. 2021. "ForSyDe-Atom: Taming Complexity in Cyber Physical System Design with Layers". *ACM Transactions on Embedded Computing Systems* 20(2):1–27.

Wadler, P., and S. Blott. 1989. "How to make ad-hoc polymorphism less ad hoc". In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 60–76. New York, NY, USA: Association for Computing Machinery.

## AUTHOR BIOGRAPHIES

**EDIL MEDEIROS** is an Adjunct Professor in the Electrical Engineering Department at University of Brasília. His research interests include continuous-time modeling for hybrid and cyber-physical systems and the compression of point clouds. His email address is j.edil@ene.unb.br.

**EDUARDO LEMOS** is a Computer Engineer focused on Functional Programming at University of Brasília. Both his final bachelor's project and master's thesis are focused on continuous-time modeling within the functional paradigm. His main work is a direct continuation of his masters advisor's, Edil Medeiros, Ph.D thesis. His email address is dudulr10@gmail.com and his homepage is https://duing.dev/.

**EDUARDO PEIXOTO** received the Engineering and M.Sc. degrees from Universidade de Brasilia, Brazil, in 2005 and 2008, respectively, and the Ph.D. degree from Queen Mary, University of London, in 2012, all in Electrical Engineering. Since 2012 he is with the Department of Electrical Engineering at Universidade de Brasília, where he is now an Adjunct Professor. His main research interests are multimedia processing and coding. His email address is eduardopeixoto@ieee.org.