An Algebra for Modeling Continuous Time Systems

José E. G. de Medeiros
Electrical Engineering Department
University of Brasília
Brasília, DF, Brazil
j.edil@ene.unb.br

George Ungureanu
Department of Electronics
KTH Royal Institute of Technology
Stockholm, Sweden
ugeorge@kth.se

Ingo Sander
Department of Electronics
KTH Royal Institute of Technology
Stockholm, Sweden
ingo@kth.se

***Abstract*—Advancements on analog integrated design have led to new possibilities for complex systems combining both continuous and discrete time modules on a signal processing chain. However, this also increases the complexity any design flow needs to address in order to describe a synergy between the two domains, as the interactions between them should be better understood. We believe that a common language for describing continuous and discrete time computations is beneficial for such a goal and a step towards it is to gain insight and describe more fundamental building blocks. In this work we present an algebra based on the General Purpose Analog Computer, a theoretical model of computation recently updated as a continuous time equivalent of the Turing Machine.**

I. Introduction

Shannon's formalization of the Sampling Theorem in 1949 established a formal model for the digital signal processing development later on. It provides a framework to convert continuous-time signals into a discrete sequence of numbers that can be processed by a digital computer. The subject of sampling can be considered a very mature discipline but some fundamental issues still remain. Aliasing is a fundamental phenomenon and some researchers propose the implementation of continuous-time circuits as a way to overcome these effects [1]. Recently, even signal processing algorithms traditionally implemented on the digital domain, such as Wavelets [2] and nonlinear differential equations solvers [3], have proposed analog/continuous-time implementations with improvements on computing speed and power consumption suggesting the emergence of hybrid analog/digital computers for embedded and cyber-physical systems (CPSs).

Despite his work on discrete time systems, Shannon also proposed in 1941 a continuous time (CT) model of computation (MoC) called the General Purpose Analog Computer (GPAC) as an effort to provide a better understanding about such systems [4]. Recently, GPAC has been updated and regarded as an equivalent of the Turing Machine in the CT domain [5]. In this work we present the GPAC MoC defined by basic units and composition rules. We present an algebra that captures the GPAC notion as a step towards the development of a simulation and synthesis language for hybrid systems.

II. GPAC: The Continuous-Time Model of Computation Formal Basis

Shannon introduced the GPAC model [4] for the Differential Analyzer, a mechanical device used to solve ordinary differential equations (ODEs) before the popularization of the digital computer. The model was further refined by Moore [6]. Graça and Costa proposed the deterministic feedforward GPAC (FF-GPAC) model by restricting the possible feedback connections on Shannon's model [7].

In the FF-GPAC, functions generated by the machine are exactly those that satisfy differential equations of the form

$$\dot{y}(t) = p(t, y(t)), \qquad t \in I \tag{1}$$

in which p is a polynomial, y is the output function, $\dot{y}$ denotes the derivative of y and I is some time interval. An FF-GPAC is capable of generating a large range of functions: rational functions (e.g. quotients of polynomials), irrational algebraic functions and algebraic-transcendental functions (e.g. exponentials, logarithms, trigonometric, Bessel, elliptic and probability functions) [4], [5]. The class of functions that are generable by Eq. (1) is closed under the usual arithmetic operations and thus one can replace p by any such generated function, e.g. $\dot{y}(t) = sin(t)$ because $sin(t)$ is in this class [8]. Besides its expressiveness power, GPAC is also prone to functionally preserving transformations [9], a topic to be further explored in the future as it may prove useful towards design automation of continuous time systems.

Moreover, the notion of complexity is being further expanded to enable comparisons between discrete and continuous time computers. Recent works show that the FF-GPAC has the same computational power as the Turing Machine [8], [9]. Still, the FF-GPAC model is founded on simple units that may be electronically implemented and that are familiar to microelectronics and control systems engineers (e.g. the continuous time computer implementation shown in [3] is based on the same basic GPAC units). We believe this to be a reasonable choice for an expressive MoC and so the FF-GPAC will be considered the basis for the discussion that follows.

A. GPAC Basic Units

The basic units define the most basic computations allowed in the GPAC. Fig. 1 shows those basic units detailed next.

a) Constant Function: A unit with constant parameter k generates a constant output $y = k$ for any time t.

b) Adder: Given two inputs u and v, it generates an output $w = u + v$, for all variations of u and v.

c) Variable Multiplier: Given two inputs u and v, it generates an output $w = uv$.

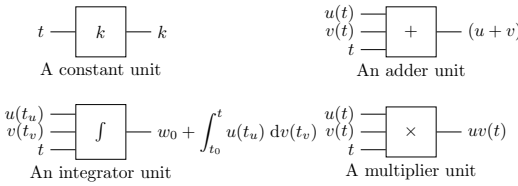

Fig. 1: GPAC basic units.

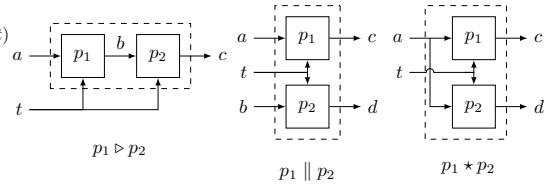

Fig. 2: GPAC composition operators

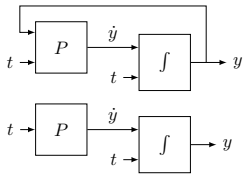

Fig. 3: Closed-loop (up) and open-loop (down) integration units

d) Integrator: Given two inputs $u(x)$ and $v(x)$, it generates an output $w(t) = w_0 + \int_{t_0}^{t} u(t_u)\mathrm{d}v(t_v)$ where w_0 is the initial setting of the integrator at t_0, u is called the *integrand* and v is called the *variable of integration*. The arguments t_u and t_v denote a concept of local time as perceived by the module that generated signals u and v respectively.

B. GPAC Composition rules

In order to construct an arbitrary deterministic function p from basic units a set of composition rules need to be enforced. First, we require that for each unit two inputs and two outputs can never be interconnected (short-circuited). Also, inputs can only be driven by either the independent variable t or by a single unit output. These are the original rules proposed by Shannon [4].

The additional restrictions from FF-GPAC [7] follow by defining acyclic (no feedback) configurations A_k to be *polynomial circuits* by using only constant function units, multipliers and adders. Thus the following conditions shall hold:

1) each input of a polynomial circuit should be the input t of the GPAC or the output of an integrator. Feedback is thus allowed only from the output of integrators to inputs of polynomial circuits.
2) each integrand input of an integrator should be driven by the output of a polynomial circuit;
3) each variable of integration of an integrator is the input t of the GPAC.

We note that the composition rules for the GPAC give rise to a regular structure as shown in Fig. 4. The polynomial circuits A_k respond instantly to changes on its inputs thus resembling combinational circuits. Integrators on the other hand are units that impose a dynamic behavior for the system in the sense that they restrict the outputs of the polynomial circuits A_k to actually behave as gradient functions dictating the output trend. For this, Graça's and Costa's FF-GPAC can be regarded as a continuous-time version of the Register-Transfer Level (RTL) abstraction in modern digital design.

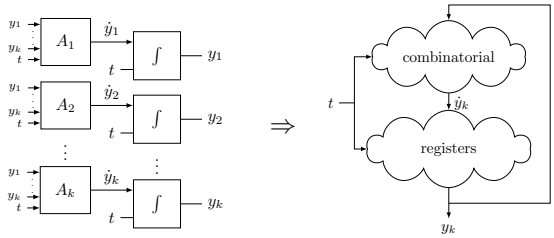

Fig. 4: A particular GPAC configuration is built as a composition of polynomial circuits A_k and integrators resembling the RTL abstraction in digital design.

III. The modeling algebra

We capture the GPAC MoC in an algebra looking forward to implement it on a domain specific language (DSL) for analysis, simulation and synthesis of continuous time hardware. As such, from this point onward with GPAC we actually denote the FF-GPAC MoC. In the definitions that follow, we call a *signal* the set of functions generated by a GPAC, and a *process* the GPAC basic units and compositions of units. We choose to describe a set of operators that capture the notion of the composition rules previously described, abstracting away the signals themselves, i.e. signals are not manipulated directly but are inferred from a composition of processes.

Interactions between the continuous time dynamics and discrete events are actually outside the scope of GPAC itself but are nevertheless essential for hybrid computers and CPSs. We propose to extend Shannon's original mechanical analogy [4] and assume that every continuous time system is confined inside a black box and the machine operator can only perform one out of two non-simultaneous operations: 1) close the black-box and let the machine execute for an arbitrary amount of time. During the execution the operator has no access to the internal state nor the outputs of the machine and the machine provides no hint about what is happening inside, it is entirely up to the operator to decide when to start and when to stop the continuous time machine; 2) open the black-box to observe its current state and reconfigure the machine if the operator decides so, i.e. to set up a new particular structure using the basic units and the composition rules discussed previously.

In a recent work [10], Lee studies the challenges of combining discrete with continuous time semantics and proposes modeling hybrid systems in a practical manner by including continuous dynamics within state machines, as *modal models*. This concept fits well with our intuition of describing GPACs as black boxes operated by "discrete" operators. As such, a GPAC could be embedded within a state machine which controls its state and inputs and collects its values, but needs not to understand the operation semantics or views of time. This paves the way for an elegant orthogonalization of different concepts of time and transition. However, a comprehensive analysis of the mechanisms of synchronization and the causality issues which arise falls out of the scope of this paper.

We capture this intuition by defining processes as recursive objects $\mathcal{P}_{CT}$ like in (8). An object of this type denotes a function that takes a time instant τ to be observed and m input values α^m and returns n values β^n corresponding to the outputs of the machine at the observation time, together with a new instance of $\mathcal{P}_{CT}$ to be executed in the future.

$$\mathcal{P}_{CT}(\alpha^m, \beta^n) : \tau \times \alpha^m \to \beta^n \times \mathcal{P}_{CT}(\alpha^m, \beta^n) \tag{8}$$

$$const_{CT} : \mathcal{P}_{CT}(\alpha, \beta) \qquad const^k_{CT} = (t, a) \mapsto (k, const^k_{CT}) \tag{2}$$

$$add_{CT} : \mathcal{P}_{CT}(\alpha \times \alpha, \alpha) \qquad add_{CT} = (t, a, b) \mapsto (a + b, add_{CT}) \tag{3}$$

$$mult_{CT} : \mathcal{P}_{CT}(\alpha \times \alpha, \alpha) \qquad mult_{CT} = (t, a, b) \mapsto (ab, mult_{CT}) \tag{4}$$

$$\triangleright : \mathcal{P}_{CT}(\alpha, \beta) \times \mathcal{P}_{CT}(\beta, \gamma) \to \mathcal{P}_{CT}(\alpha, \gamma) \qquad p_1 \triangleright p_2 = (t, a) \mapsto (c, p_1' \triangleright p_2') \quad \text{where} \quad (b, p_1') = p_1(t, a),\ (c, p_2') = p_2(t, b) \tag{5}$$

$$\parallel : \mathcal{P}_{CT}(\alpha, \beta) \times \mathcal{P}_{CT}(\gamma, \delta) \to \mathcal{P}_{CT}(\alpha \times \beta, \gamma \times \delta) \qquad p_1 \parallel p_2 = (t, a) \mapsto (c, d, p_1' \parallel p_2') \quad \text{where} \quad (b, p_1') = p_1(t, a),\ (c, p_2') = p_2(t, b) \tag{6}$$

$$\star : \mathcal{P}_{CT}(\alpha, \beta) \times \mathcal{P}_{CT}(\alpha, \gamma) \to \mathcal{P}_{CT}(\alpha, \beta \times \gamma) \qquad p_1 \star p_2 = (t, a) \mapsto (b, c, p_1' \star p_2') \quad \text{where} \quad (b, p_1') = p_1(t, a),\ (c, p_2') = p_2(t, b) \tag{7}$$

We then define the three stateless GPAC basic units, represented in Fig. 1. $const^k_{CT}$ in (2) is a function that takes an observation time t and an input a and outputs a constant value k. $adder_{CT}$ in (3) and $mult_{CT}$ in (4) take an observation time t and two inputs a and b and output $a + b$, respectively $a \times b$.

Next we introduce three operators to capture GPAC composition rules for describing polynomial circuits depicted in Fig. 2. The cascade operator $\triangleright$ in (5) captures the notion of serial composition: $p_1 \triangleright p_2$ is a process that takes an input and evaluates both processes at time t, using the output of p_1 as the input to p_2. The parallel operator $\parallel$ in (6) abstracts parallel composition: $p_1 \parallel p_2$ is a process that takes two inputs and evaluate both processes at time t generating two outputs. Finally, the fan-out operator $\star$ in (7) captures the notion of a single input feeding two parallel processes: $p_1 \star p_2$ is a process that takes a single input and evaluates both processes at time t to generate two outputs. The basic objects and operators defined so far are sufficient to describe polynomial GPAC circuits while ensuring that the machine structure does not change in between function evaluations at different observation times.

A. Integration and Feedback

Eq. (1) suggests an interesting fact, generally overlooked: feedback is not a mere topological option for the designer but is in fact an essential part of continuous time systems specification. Consider the open loop GPAC shown in Fig. 3. The open loop configuration leads to the system in (9), i.e., the only possible generable functions on an open loop configuration are polynomial functions. Feedback, on the other hand, leads to the more general family of functions described by Eq. (1).

$$\begin{aligned} \dot{y}(t) &= p(t), \quad t \in I \\ y(t) &= \int_I p(t)\ \mathrm{d}t, \end{aligned} \tag{9}$$

$$\begin{cases} \dot{y}(t) = z(t) \\ \dot{z}(t) = -y(t) \end{cases} \tag{10}$$

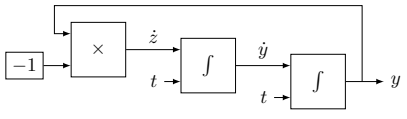

Fig. 5: GPAC configuration that generates $y(t) = \sin(t)$.

A problem involving ODEs however, is not completely specified by its equations only. Consider the sine wave generator described by the GPAC in Fig. 5 [8] defined as the system of equations (10). The general solution for this circuit is $y(t) = k \sin(t + \theta)$ and $z(t) = k \cos(t + \theta)$, $\forall\ k, \theta \in \mathbb{R}$. The verification of this result is trivial by substituting it in Eqs. (10) and is left to the reader. As for the same system several outputs are equally valid, the system (10) itself constitutes a nondeterministic model. A set of initial conditions is thus the information that collapses this nondeterministic model into a deterministic one. In fact, the knowledge of any pair of conditions $y(t_0) = y_0$, $z(t_0) = z_o$ completely determines the system.

As of GPAC composition rules, feedback is only allowed from an integrator unit to a polynomial (stateless) circuit. Therefore we introduce a feedback operator $\oint$ that captures this notion in Eq. (11), instead of an open-loop block like in Fig. 1. The operator $\oint$ takes a CT process p_1 with two inputs and transforms it into a single input process with the closed-loop feedback structure shown in Fig. 3. The parameters t_0 and y_0 capture the initial conditions as part of the system specification. In the definition of the feedback operator we assume that time can only advance, i.e. $t \geq t_0$, and thus the initial conditions of the process generated by observing the system encode the history of that system.

$$\begin{aligned} \oint &: \mathcal{P}_{CT}(\alpha \times \beta, \gamma) \to \mathcal{P}_{CT}(\alpha, \gamma) \\ \oint_{t_0, y_0} p_1 &= (t, a) \mapsto (c, \oint_{t,d} p_1') \end{aligned} \tag{11}$$

where $c = y(t)$ such that $y(t)$ satisfies $\dot{y}(x) = p_1(x, a, y(x))$ with $y(t_0) = y_0$ at time t, d is an arbitrary number and p_1' an arbitrary new process. On the new process generated by evaluating $\oint_{t_0, y_0} p_1$ we enforce time continuity by requiring a new initial condition pair to be (t, d), i.e. the new initial condition can be any arbitrary state d but must be defined at the observation time t. In the particular case in which $c = d$, i.e., the new initial condition is the output of the machine at the observation time t, continuity on the output is guaranteed. This setup is purely denotational, giving no hint about how to actually solve the ODE. In Section IV we discuss one approach to embed ODE solvers into an executable language aiming simulation of such systems.

This setup provides the advantage of being sufficiently expressive to model open-loop integrators out of closed-loop ones by using the structure described by (12), where p_1 is an arbitrary GPAC and id is an identity process that propagates its input at an observation time t to the output. This structure simply implements equation $\dot{y}(t) = p_1(t) + 0 \times y(t) = p_1(t)$ which is equivalent to (9).

$$\begin{cases} y = \oint_{t_0, y_0} (p_1 \parallel loopBreaker) \triangleright add_{CT} \\ loopBreaker = \left(const^0_{CT} \star id\right) \triangleright mult_{CT} \end{cases} \tag{12}$$

IV. Simulation

As a proof of concept, we developed an embedded domain specific language (EDSL) on the functional language Haskell that implements this algebra[1]. We briefly present two key concepts we rely on: lazy evaluation and explicit ODE solvers.

[1]EDSL and experiments found at `https://github.com/forsyde/reactive-gpac`

GPAC	EDSL	Comments	GPAC	EDSL	Comments
$\mathcal{P}_{CT}$	`PCT`	constructor `prCT`	τ	`Time`	host type for time
$const^{k}_{CT}$	`constCT k`		$\oint$	`intCT`	solver as argument
add_{CT}	`addCT`		$\triangleright$	`>>>`	infix operator
mul_{CT}	`mulCT`		$\|\|$	`***`	infix operator
id_{CT}	`idCT`		$\star$	`&&&`	infix operator

TABLE I: GPAC algebra as EDSL in Haskell.

Lazy evaluation [11] is an strategy which delays the evaluation of an expression until its value is needed. One of its useful features is the ability to propagate unknowns as a chain of operations throughout a program, enabling the definition of control flow as abstractions instead of primitives. This strategy permits the abstraction of processes $\mathcal{P}_{CT}$ in a continuation style data type in which the results of a computation bring a new function to be evaluated in the next computation round. This translates into a potentially infinite structure of discrete processes to be evaluated.

The denotational nature of the feedback operator $\oint$, Eq. (11) implies that there are potentially many possible implementations for such algebra. For this paper we chose to instantiate the operator $\oint$ by using a collection of explicit one-step ODE solvers, i.e., algorithms in which $y(t) = \Phi(t_0, y_0, t, f)$, i.e., $y(t)$ is approximated by a function that depends on the initial condition, the observation time and the system topology. Explicit Runge-Kutta methods are a family of such algorithms [12] and are used.

We introduce the semantic function `at` to the language in which `p 'at' t` takes a top-level process p and observes its output on the specified time t. In this way, we introduce two notions of time in the framework: 1) the *local time* as seen by each process as the result of its management of time advancement for a certain computation; 2) the *observation time*, or *global time*, as seen by the GPAC operator and communicated to the system via the `at` function.

```
at :: PCT () a -> Time -> a
p `at` t = pT
  where (pT, _) = prCT p t ()
```

We describe the GPAC of Fig. 5 and initial conditions $z(0) = 1$, $y(0) = 0$ in our EDSL with the function `sineGPAC`, which implements (10) using the open-loop integrator defined in (12). For simulation we define three processes $\texttt{p1}(t) = \sin(t)$, $\texttt{p2}(t) = \sin(2t)$ and $\texttt{p3}(t) = \sin(t + \pi/2)$. The processes `tScale` and `tShift` distort the local time as seen by `sineGPAC`. One can prove, e.g., that `time >>> tScale 2 >>> sineGPAC` $\equiv$ `time >>> sineGPAC (\t -> 2*t)` by algebraically applying the composition operators definitions. Fig. 6 shows the output of these processes as observed at the same time instants. These examples also show how to introduce hierarchy on the models by using the keyword `where` to hide a process inside another one.

```
sineGPAC = intCT rk4 0 0 p1
  where
    p1 = (constCT (-1) *** idCT) >>> multCT >>> integrator
    integrator  = intCT rk4 0 1 loopBreaker
    loopBreaker = (idCT *** constCT 0) >>> adderCT
```

```
tScale k = (idCT &&& constCT k) >>> multCT
tShift k = (idCT &&& constCT k) >>> adderCT

p1 = time >>> sineGPAC
p2 = time >>> tScale 2 >>> sineGPAC
p3 = time >>> tShift (pi/2) >>> sineGPAC
```

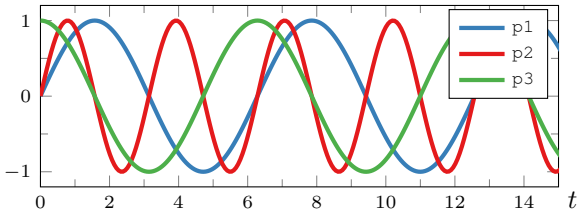

Fig. 6: Local time distortion.

V. Conclusions & Future Work

We define an algebra with basic symbols and composition rules for a CT MoC. We propose modeling integration only in the context of feedback, contrary to all other languages we are aware of. As discussed, this approach enforces closer agreement to the formal model of computation which is tinkered to avoid giving rise to non-deterministic systems. Finally, we show a work-in-progress EDSL implementing the proposed algebra which we believe is an important step towards simulation, formal verification and synthesis of hybrid systems.

VI. Acknowledgement

This work was partially supported by CNPq, the Brazilian National Scientific Development Council; by CISB, the Swedish-Brazilian Research and Innovation Office; and by Saab AB.

References

[1] B. Schell and Y. Tsividis, "Analysis of Continuous-Time Digital Signal Processors," in *2007 IEEE Intern. Sympos. on Circuits and Systems (ISCAS2007)*, May 2007, pp. 2232–2235.

[2] J. M. H. Karel *et al.*, "Implementing Wavelets in Continuous-Time Analog Circuits With Dynamic Range Optimization," *IEEE Trans. Circuits Syst. I*, vol. 59, no. 2, pp. 229–242, Feb. 2012.

[3] N. Guo *et al.*, "Energy-Efficient Hybrid Analog/Digital Approximate Computation in Continuous Time," *IEEE J. Solid-State Circuits*, vol. 51, no. 7, pp. 1514–1524, Jul. 2016.

[4] C. E. Shannon, "Mathematical Theory of the Differential Analyzer," *Journal of Math. and Phys.*, vol. 20, no. 1-4, pp. 337–354, Apr. 1941.

[5] D. Silva Graça, "Some recent developments on Shannon's General Purpose Analog Computer," *Math. Logic Quarterly*, vol. 50, no. 4-5, pp. 473–485, Sep. 2004.

[6] C. Moore, "Recursion theory on the reals and continuous-time computation," *Theor. Computer Science*, vol. 162, no. 1, pp. 23–44, Aug. 1996.

[7] D. Silva Graça and J. Félix Costa, "Analog computers and recursive functions over the reals," *Jour. of Complexity*, vol. 19, no. 5, pp. 644–664, Oct. 2003.

[8] O. Bournez, D. Graça, and A. Pouly, "On the Functions Generated by the General Purpose Analog Computer," *arXiv:1602.00546*, Jan. 2016.

[9] A. Pouly, "Continuous models of computation: from computability to complexity," Ph.D. dissertation, Ecole Doctorale Polytechnique, France; Universidad do Algarve, Portugal, Jul. 2015.

[10] E. A. Lee, "Fundamental Limits of Cyber-Physical Systems Modeling," *ACM Trans. Cyber-Phys. Syst.*, vol. 1, no. 1, pp. 3:1–3:26, Nov. 2016.

[11] S. L. Peyton Jones, *The Implementation of Functional Programming Languages*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987.

[12] J. Stoer and R. Bulirsch, *Introduction to numerical analysis*. Springer Science & Business Media, 2013, vol. 12.