# A Layered Formal Framework for Modeling of Cyber-Physical Systems

George Ungureanu and Ingo Sander
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden
Email:{*ugeorge*, *ingo*}@kth.se

***Abstract*—Designing cyber-physical systems is highly challenging due to its manifold interdependent aspects such as composition, timing, synchronization and behavior. Several formal models exist for description and analysis of these aspects, but they focus mainly on a single or only a few system properties. We propose a formal composable framework which tackles these concerns in isolation, while capturing interaction between them as a single *layered model*. This yields a holistic, fine-grained, hierarchical and structured view of a cyber-physical system. We demonstrate the various benefits for modeling, analysis and synthesis through a typical example.**

## I. Introduction

The *orthogonalization of concerns*, i.e. the separation of the various aspects of design, as already pointed out in 2000 by Keutzer et al. [1], is an essential component in a system design paradigm that allows a more effective exploration of alternative solutions. This paper aims at pushing this concept to its limits, by not only literally separating concerns and describing simple rules of interaction, but also deconstructing them to their most *basic*, *composable* semantics.

We present a pure functional framework for describing, analyzing and simulating cyber-physical systems (CPS) as *networks of processes* communicating through *signals* similar to [2]–[8]. The purity inherent from describing behavior as *side-effect free functions* ensures that each and every entity is independent and contains self-sufficient semantics, enabling both the modeling and the simulation of *true concurrency* without the need of a global state or controller. Binding together concepts from a wide pool of formalisms this *unified* framework exhibits three essential properties: 1) it slices processes in *structured enclosing layers* abstracting different behavioural aspects, as an embodiment of separation of concerns; 2) it provides *primitive, indivisible building blocks* for each layer, named *atoms*, as the outcome of deconstructing the abstract semantics to their core; 3) it describes composition in terms of *patterns* of building blocks, as a means of both modeling complex behavior within a layer, as well as abstracting it in a hierarchical manner. We demonstrate the potential for modeling, analysis and synthesis through a comprehensible example.

## II. Background

The framework is built on three key foundations: the tagged signal model, the functional programming paradigm and algorithmic skeletons. This section introduces a few notions to prepare the reader for the discussion of the layered process model in Section III.

*1) The tagged signal model:* is a denotational framework introduced by Lee and Sangiovanni-Vincentelli [2] as a common meta model for describing properties of concurrent systems in general terms as sets of possible behaviors. Systems are regarded as *compositions* of *processes* acting on *signals*. Signals are collections of events and are characterized by a *tag system*. The tag system describes causality between events, and can model time, precedence relationships, synchronization points, and other key properties. Based on how tag systems are defined, one can identify several *models of computation (MoCs)* as classes of behaviors dictating the semantics of execution and concurrency in a network of processes.

A signal is described as a set of events, where each event $e_i$ is composed of a tag $t_i \in T$ and a value $v_i \in V$ (1). From this definition we can further classify MoCs as: *timed* where the set of tags $T$ is a total order (e.g. continuous time – CT, discrete event – DE, synchronous – SY); or *untimed*, where $T$ is a partial order (e.g. synchronous data flow – SDF). A process can be described as either a set of possible behaviors of signals or as a relation between multiple signals. Complex systems are modeled by composing processes, which in this case is interpreted as the "intersection of the behaviors of each of the processes being involved" [2]. Functional frameworks [4]–[6], define processes as *functions* on the (history of) signals (2), whereas complex behavior is modeled by means of *process composition*. According to this view, arbitrary compositions of processes can be regarded as networks of processes.

$$\begin{aligned} &s = \{e_0, e_1, ...\} \in S \\ &\text{where } e_j = (t_j, v_j) \\ &\qquad v_j \in V,\ t_j \in T \\ &\qquad t_j \leq t_{j+1},\ \forall\ j \in \mathbb{N} \end{aligned} \tag{1}$$

$$p : S^m \rightarrow S^n \qquad p(s^m) = (s^n) \tag{2}$$

$$f : V^m \rightarrow V^n \in V \tag{3}$$

*2) The functional programming paradigm:* treats computation as evaluations of side-effect-free mathematical functions and offers a broad set of formal tools for reasoning about the correctness of a program, e.g. proofs-as-programs, type checking or equational reasoning. A key concept adopted in this paper is the notion of *higher-order functions*. These are functions that take functions as arguments and return functions as results. Since functions are treated as first-class citizens, the set of values $V$ may include functions (3), enabling to treat a function as any other value.

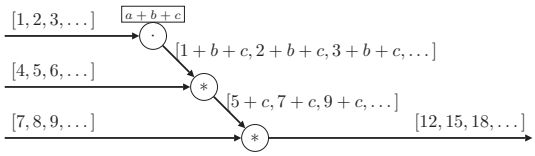
978-3-9815370-8-6/17/$31.00 

*Partial application* is a mechanism based on currying which allows to take functions with arbitrary number of arguments and systematically apply one argument at a time, yielding intermediate functions with fewer arguments. For example, in (4), taking function $f$ and fixing one argument $a = 1$ yields function $g$.

$$\begin{array}{l} f : \alpha \to \overbrace{\beta \to \gamma}^{g} \\ f\ a\ b = a + b \end{array} \overset{a=1}{\mapsto} \begin{array}{l} g : \beta \to \gamma \\ g\ b = (f\ 1)\ b = 1 + b \end{array} \tag{4}$$

*Structured data types* are types built using type constructors (e.g. $\mathcal{C}$) thus exposing certain properties. The idea of building structures around other structures which enable computational behavior sets up the foundation of the layered process model.

A central concept in defining atoms is the *applicative functor* [9]. A functor is a higher-order function which takes a function $f : \alpha \to \beta$ and applies it to each element in a structure $\mathcal{C}(\alpha)$ yielding a structure $\mathcal{C}(\beta)$. An applicative functor on the other hand, takes a structure of functions $\mathcal{C}(\alpha \to \beta)$ and applies it on $\mathcal{C}(\alpha)$ yielding $\mathcal{C}(\beta)$. Serializing these functions enables a systematic partial application of a combinatorial function with an arbitrary number of arguments $f : \alpha^n \to \beta$ on an arbitrary number of structures $\mathcal{C}(\alpha)^n$. Below we depict an example composition of a functor '$\cdot$' with an applicative functor '$*$' to merge and add elements of three lists, where a list is a structured data type.




*3) Algorithmic skeletons:* Cole coined this term [10] to denote high-level constructs encapsulating parallelism, communication and synchronization with an associated cost complexity. The key idea behind algorithmic skeletons is that there exists a set of functions on structured data types called *homomorphisms* which respect the way in which objects of their type are built [11], [12]. Intuitively, homomorphisms enable computation on separate parts of a large data structure to be performed in parallel.

## III. A Layered Process Model

We introduce a framework that binds the previously presented formal concepts in a structured fashion, in order to describe platform-independent, analyzable and synthesizable cyber-physical systems (CPS). This model relies on the premise that *systems can be described as side-effect-free functions over the spatial and temporal dimensions of data*, i.e. systems are driven by data, or more precisely, by the way the data is constructed. Throughout the development of this framework we were led by two policies: 1) it is of utmost importance to separate the different concerns of CPS in order to cope with their complexity; 2) we aim to provide primitive indivisible building blocks in order to have a small, ideally minimal, grammar to reason with and to express complex aspects of system behavior through means of composition and generalization.

The first policy led to what we call a *layered process model*, reflecting the need to have a clear view on the modeling and analysis of different aspects of a system. We have identified four main layers as depicted in Fig. 1. The innermost layer consists simply of operations on data exposing a system's functionality, whereas each higher layer abstracts a separate aspect and operates on a different structure. Each layer is implemented as a set of *higher-order functions* where layer $L$ takes functions of layer $L-1$ as arguments. These higher-order functions are called *constructors*, and are provided as building blocks that abstract certain behaviors. Constructors need to have meaningful semantics, i.e. should be analyzable, and can be associated with a target platform implementation, i.e. should be synthesizable. Complex behaviors can be achieved by means of arbitrary *compositions* of the provided constructors.

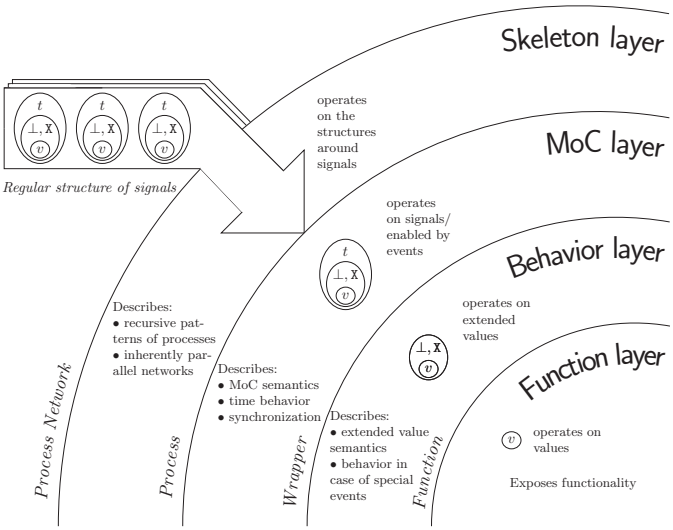



Fig. 1: The layered process model

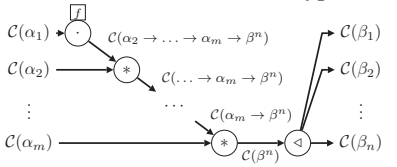
### A. Atoms, Constructors and Patterns

The idea of a finite set of higher-order functions abstracting formal semantics has been valued across research fields [3]–[8], [13]–[17] providing elegant solutions. For example, ForSyDe defines *process constructors* as key building blocks for enabling both formal reasoning and design transformation [5]. Driven by this, we extend the use of higher-order functions as constructors not only for processes inside the MoC layer, but for entities associated with all layers.

The second policy motivated us to synthesize the most basic composable operations holding meaningful semantics for a specific layer and provide them as primitive constructors called *atoms*. This allows to define a *minimalistic layered formal framework* for describing CPS that: 1) conveys a fine-grained and analyzable view of a heterogeneous system's functionality; 2) abstracts behavior through a set of formal rules; 3) scales and is extended by means of systematic composition of atoms. We use the term *patterns* to denote constructors expressing a meaningful composition of atoms, to distinguish them from primitive constructors.

The main formal reasoning behind atoms derives from the theory of *applicative functors* [9]. Among others, each layer defines a pair of applicative atoms as infix operators, e.g. $(\cdot, *)$,

which, aided by a *type traversal* [18] utility '$\triangleleft$', allows to form $m$-input and $n$-output entities in the sense of (2). The implied composition of a generic pattern which maps function $f$ on a series of structures $\mathcal{C}$, as a typical constructor, is shown in (5).




$$\begin{aligned} &\texttt{cons}_L : (\alpha^m \to \beta^n) \\ &\quad \to \mathcal{C}(\alpha)^m \to \mathcal{C}(\beta)^n \\ &\texttt{cons}_L(f)(s^m) = f \cdot s_1 \\ &\quad * s_2 * \ldots * s_m \triangleleft_n \end{aligned} \tag{5}$$

Throughout this paper we identify constructors using the naming convention below. The following sections offer an overview for the presented layers, whereas detailed definitions can be found in the online documentation at [19].

| Layer | infix operator | example | prefix name | example |
|---|---|---|---|---|
| Behavior | $\square$ | $\boxdot$,$\circledast$ | $_b$ | $\texttt{default}_b$ |
| MoC | $\bigcirc$ | $\odot$,$\circledast$ | $_p$ | $\texttt{comb}_p$ |
| Skeleton | $\diamond$ | $\diamondsuit$,$\diamondsuit$ | $_s$ | $\texttt{map}_s$ |

### B. The Function Layer

The bottom layer is exposing the operations on data performed by a system and is as complete or as expressive as the host language defines it. To be consistent with our framework, we need to constrain operations in this layer to be formulated as side-effect-free functions on values (i.e. $V^m \to V^n$), and to properly respect the interfaces to the behavior layer above.

### C. The Behavior Layer

This layer addresses the fact that CPS often deal with special events which cannot be represented by any set of ordinary values. For example, the family of synchronous languages [20] denote the absence of an event with a symbol $\perp$. Another example, the `std_logic` data type in VHDL uses nine different values for binary logic, e.g. `X` for unknown values. These values requires a resolution function to mirror the behavior of the underlying technology.

Inspired by [2], we represent these events by extending the value set $V$ in (1) with tokens carrying special semantics and inferring a specific behavior. It is important to view the set of extended values as a structure $\mathcal{B}(V)$ enabling the atoms to capture the specific behavior, but in our notations we use the shorthand $V_e$.

Since the nature of this layer is isomorphic to a platform model or technology, we can argue the existence of classes of behaviors rather than a universal behavior model. Common to most classes are the atoms: '$\boxdot$' grants a default behavior to a function based on the definition of the special event (6a); '$\circledast$' offers a resolution between two extended values (6b); '$\boxplus$' replaces an extended value with another one based on a boolean predicate, and it is commonly used to model filtering behaviors (6c).

$$\boxdot : (V \to V) \to (V_e \to V_e) \tag{6a}$$
$$\circledast : V_e \to V_e \to V_e \tag{6b}$$
$$\boxplus : V_e \times \mathit{Bool} \to (V_e \to V_e) \tag{6c}$$

An important pattern in this layer is $m$-input and $n$-output behavior wrapper $\texttt{default}_b$ (infix '$\boxplus$') which specifies a default resolution between multiple extended values. It is defined as the composition (5) of the atoms (6a) and (6b):

$$\boxplus : (V^m \to V^n) \to (V_e^m \to V_e^n) \tag{7}$$

### D. The MoC Layer

The MoC layer atoms embed timing and synchronization semantics as dictated by a specific MoC. Our framework encodes a MoC's tag system (1) in the structure of events $\mathcal{E}(V_e) = T \times V_e \in E$. Consequently MoC atoms are enabled by events, or more precisely by their structure.

Recalling the definition of signals as ordered sets of events (1), and of processes as functions (2) we can describe processes in the sense of [2] as entities constructed with MoC atoms. Consequently, signals are also structures $\mathcal{S}(E) \in S$. The most natural way to express signals was initially formulated by Reekie's *streams* [3] and followed by others [4]–[8], namely as infinite lists [21]. This model allows the description and simulation of systems as *data flow networks of processes*.

We define a set of four atoms as functional descriptions of the main properties of processes in [2]: '$\odot$' confers a possible *behavior* to a process, by mapping a function of extended values on all the events of an input signal (8a); '$\circledast$' establishes a *relation* between two signals (8b). It practically synchronizes two signals and applies the functions carried by one signal on the arguments carried by the other, outputting a third signal carrying the synchronized results; '$\circledcirc$' ensures *complete partial order* in signals, by creating and pre-pending an initial event to a signal (8c); '$\circledcirc$' allows the manipulation of tags in a restrictive way which preserves *monotonicity and continuity* in a process, namely by "phase-shifting" all tags in a signal with a constant (8d). In (8a) we use the $\vdash$ symbol to separate the function passed to the '$\odot$' atom, from an optional context $\Gamma$ used to capture various execution parameters under which $V_e \to V$ is evaluated, e.g. rates in SDF processes.

$$\odot : (\Gamma \vdash V_e \to V) \to (S \to S) \tag{8a}$$
$$\circledast : S \to S \to S \tag{8b}$$
$$\circledcirc : V_e \to S \to S \tag{8c}$$
$$\circledcirc : T \to S \to S \tag{8d}$$

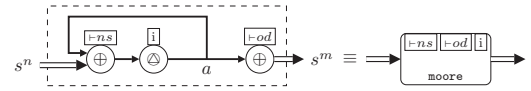
Formulae (8) describe only the atoms' type signatures as interfaces respected by all MoCs. In other words each supported MoC overloads these operators with its own execution semantics. This indicates that the composition of atoms is *completely decoupled* from the semantics abstracted by the atoms themselves. This property is demonstrated in Section IV by showing that the same process network behaves differently depending on the tag system, i.e. MoC, it operates on.

We can recapitulate that as concerning the timing aspects in a CPS, we describe a minimal set of building blocks, an obvious benefit in both reasoning and formal proofs. Using this grammar we are able to model components commonly used in CPS as specific compositions, irrespective of the execution semantics, hence justifying the name of "patterns". Among them we highlight the following: the $m$-input and $n$-output combinational process constructor $\texttt{comb}_p$ (infix '$\oplus$') (9a); the $\texttt{delay}_p$ element (infix '$\triangle$') whose execution semantics can be deconstructed into a pre-pend and a phase-shift (9b); a Moore state machine, where $ns$ is a next-state function, $od$ is the output decoder, $i$ is an initial state (9c).

$$\oplus : (\Gamma \vdash V_e^m \to V_e^n) \to (S^m \to S^n) \qquad (9a)$$
$$\vdash f_b \oplus s^m = \vdash f_b \odot s_1 \circledast \ldots \circledast s_m \lhd_n$$

$$\triangleleft : E \to S \to S \qquad (9b)$$
$$(t, v) \triangleleft s = t \circledcirc v \oslash s$$

$$\texttt{moore}_p : (\Gamma \vdash V_e \times V_e^n \to V_e) \times (\Gamma \vdash V_e \to V_e^m) \times E \to (S^n \to S^m) \qquad (9c)$$
$$\texttt{moore}_p(\vdash ns, \vdash od, i)(s^n) = \vdash od \oplus a$$
$$\text{where } a = i \triangleleft \vdash ns \oplus (a, s^n)$$




The drawing above depicts the pattern in (9c) as a process network. While the left hand side depicts the well-recognizable pattern of a Moore machine, the right hand side presents it as a primitive in a possibly larger process network, abstracting the inferred composition. This shows the possibility to provide patterns as hierarchical components which can be plugged in through the same composition rules. The generality of this principle is ensured by the minimality of the underlying grammar, i.e. functions and compositions of functions. In conclusion, each layer can be naturally augmented with a second dimension of practically unlimited levels of hierarchy, enabling our framework to also be regarded from a component-based design perspective.

### *E. The Skeleton Layer*

This layer is a natural extension which fits perfectly with our view of structured data-driven systems. If we consider signals wrapped in regular structures $\mathcal{H}(S)$ (short-hand $\langle S \rangle$), then we can formulate homomorphisms as atoms on those structures, taking processes as arguments. As a consequence those atoms will instantiate regular networks of processes which expose the *inherent parallelism*, concluding that the atoms specific to this layer are indeed algorithmic skeletons [12].

Skillicorn has proved in [11] that all algorithmic skeletons can be expressed as compositions of two primitives: *map* and *reduce*. Hence, by describing these two primitives we can claim complete minimality for the skeleton layer. Our framework provides the following atoms: '$\diamondsuit$' maps a process on a structure of signals (10a); '$\circledast$' applies a structure containing processes on a structure containing signals (10b); $\texttt{reduce}_s$ (infix '$\diamondsuit$') reduces a structure of signals to one signal based on binary process. (10c).

$$\diamondsuit : (S \to S) \to \langle S \rangle \to \langle S \rangle \qquad (10a)$$
$$\circledast : \langle S \to S \rangle \to \langle S \rangle \to \langle S \rangle \qquad (10b)$$
$$\diamondsuit : (S \to S \to S) \to \langle S \rangle \to S \qquad (10c)$$

A multi-argument $\texttt{map}_s : (S^m \to S^n) \to \langle S \rangle^m \to \langle S \rangle^n$ (infix '$\oplus$') is expressed following the composition rule (5) on the atoms '$\diamondsuit$' and '$\circledast$'. The atom overloading concept from (8) applies for (10) when considering different structures, e.g. vectors, lists, trees.

The regular structures $\mathcal{H}$ are in the programming literature often referred to as categorical types [11], [22] and are found in frameworks concerning massively parallel computing like [13]–[17]. There is a plethora of process networks that can be created using algorithmic skeletons, varying from process farms and reduction networks to systolic arrays and meshes. For example, [3] shows how elegantly the butterfly network of an FFT can be described using merely skeleton primitives.

The MoC and the skeleton layers can be regarded as being complementary. Since both $\mathcal{S}$ (i.e. infinite lists) and $\mathcal{H}$ are categorical types, they are both being backed up by the Bird-Merteens formalism [22] which enables derivation of systems from specification to implementation by means of equational reasoning [3], [5], [13]. However, these two layers have a different regard to the partitioning of data: while $\mathcal{S}$ describes partitioning in time, $\mathcal{H}$ is concerned with spatial partitioning. In this sense our framework is able to describe both true concurrency as processes with independent time models, but also massive parallelism as regular process networks.

## IV. Views and Projections – Case Study

In this section we present a synthetic process network in order to visualize the elegant simplicity yet great potential of the proposed layered compositional framework. After describing the application in terms of constructors, we discuss some projections of the constructed patterns on each layer, from top to bottom, suggesting a potential synthesis flow. Lastly we observe the system's response to signals of different MoCs.

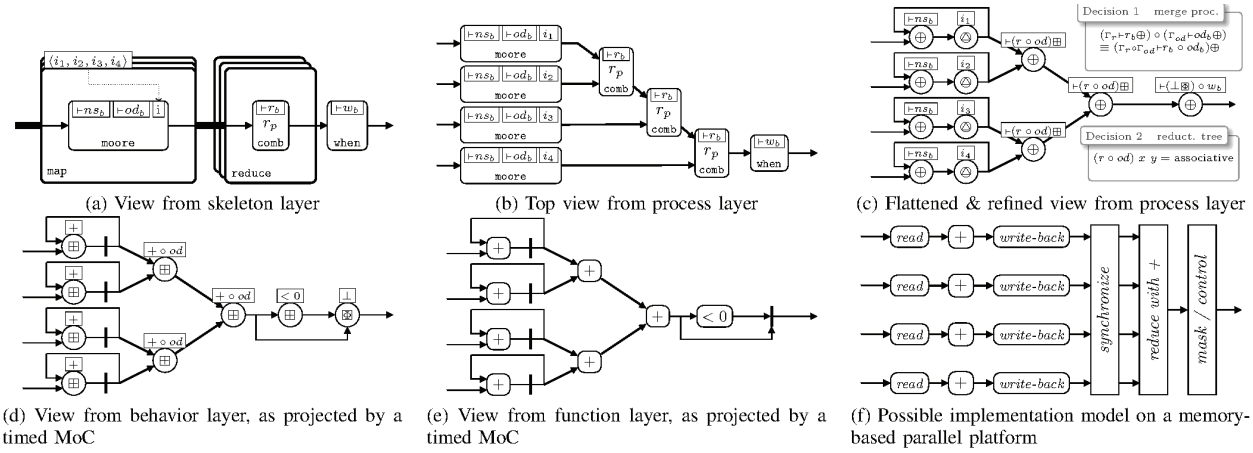
Using the atoms and patterns introduced in Section III, we can describe a system `sys` (11) where '$\circ$' is the function composition operator and the vector of initial states $\langle i \rangle$ along with the contexts and functions for next state $\Gamma_{ns} \vdash ns_b$, output decoder $\Gamma_{od} \vdash od_b$, reduction $\Gamma_r \vdash r_b$ and predicate $\Gamma_w \vdash w_b$ are given in Table I. As mentioned in Section III-D, composition is static and independent from the MoC semantics hosted by atoms. On the other hand the arguments passed to the process constructors reflect the way a MoC is built. E.g the initial states $i_j$ in the Moore process expose their tag system.

$$\texttt{sys} : \langle V \rangle \to \langle S \rangle \to S \qquad (11)$$
$$\texttt{sys}\langle i \rangle \langle s \rangle = \texttt{when}_p(\Gamma_w \vdash w_b) \circ \texttt{reduce}_s(r_p) \circ \texttt{map}_s(pc_p)\langle i \rangle \langle s \rangle$$

where
$$\texttt{when}_p(\Gamma_w \vdash w_b)(s) = ((\bot \boxplus) \circ (\Gamma_w \vdash w_b)) \oplus s \qquad (11a)$$
$$r_p(x, y) = \Gamma_r \vdash r_b \oplus (x, y) \qquad (11b)$$
$$\texttt{map}_s(pc_p)\langle v \rangle \langle s \rangle = pc_p \diamondsuit \langle v \rangle \circledast \langle s \rangle \qquad (11c)$$
$$pc_p(x, y) = \texttt{moore}_p(\Gamma_{ns} \vdash ns_b, \Gamma_{od} \vdash od_b, x)(y) \qquad (11d)$$

Fig. 2a depicts the top-level view of (11) projected onto the skeleton layer, where the skeletons $\texttt{map}_s$ and $\texttt{reduce}_s$ are represented as shaped clusters around processes, and $\texttt{when}_p$ (11a) filters a signal similarly to the `when` construct in Lustre [20]. The usage of $\texttt{map}_s$ reveals the power of partial application in combination with applicative functors. In the explicit definition in (11c), we are passing a partially applied $\texttt{moore}_p$ process constructor and a vector of initial states instead of a process instance. In fact this enables to instantiate a farm of Moore machines, each one having a different initial state.

Unfolding $\texttt{map}_s$ and $\texttt{reduce}_s$ grants the system projection onto the MoC layer from Fig. 2b. Even more, by flattening the hierarchy within this layer, we obtain a more fine-grained view which may expose further potential for optimization. E.g. based on a cost analysis, the output decoder from the Moore pattern can be merged with $r_p$, like in Fig. 2c. Also, observing that the function in $r_p$ is associative enables the transformation of $\texttt{reduce}_s$ into a $O(\log n)$ reduction tree [11].




(a) View from skeleton layer (b) Top view from process layer (c) Flattened & refined view from process layer

(d) View from behavior layer, as projected by a timed MoC (e) View from function layer, as projected by a timed MoC (f) Possible implementation model on a memory-based parallel platform

Fig. 2: Views and projections for the example system described in Eq. (11)

TABLE I: CONTEXTS, FUNCTIONS AND INITIAL TOKENS FOR THE SYSTEM IN EQ. (11) AND FIG. 2

| MoC | $\Gamma_w \vdash$ $w_b(x)$ | $\Gamma_r \vdash$ $r_b(x,y)$ | $\Gamma_{ns} \vdash$ $ns_b(x,y)$ | $\Gamma_{od} \vdash od_b(x)$ | $\langle i \rangle = \langle (t,v) \rangle$ |
|---|---|---|---|---|---|
| SDF[‡] | $2,2 \vdash (x_1 < 0, x_2 < 0)\boxplus$ | $(1,1),1 \vdash (x_1 + y_1)\boxplus$ | $(1,2),1 \vdash (x_1 + y_1 + y_2)\boxplus$ | $1,1 \vdash x_1\boxplus$ | $\langle$ $(\ ,-1)$ $(\ ,1)$ $(\ ,-1)$ $(\ ,1)$ $\rangle$ |
| SY | $\vdash (x < 0)\boxplus$ | $\vdash (x + y)\boxplus$ | $\vdash (x + y)\boxplus$ | $\vdash x\boxplus$ | $\langle$ $(\ ,-1)$ $(\ ,1)$ $(\ ,-1)$ $(\ ,1)$ $\rangle$ |
| DE | $\vdash (x < 0)\boxplus$ | $\vdash (x + y)\boxplus$ | $\vdash (x + y)\boxplus$ | $\vdash x\boxplus$ | $\langle$ $(5,-1)$ $(14,1)$ $(10,-1)$ $(14,1)$ $\rangle$ |
| CT | $\vdash (x < 0)\boxplus$ | $\vdash (x + y)\boxplus$ | $\vdash (x + y)\boxplus$ | $\vdash x\boxplus$ | $\langle (.05, \lambda t \to -1)$ $(.14, \lambda t \to 1)$ $(.1, \lambda t \to -1)$ $(.14, \lambda t \to 1) \rangle$ |

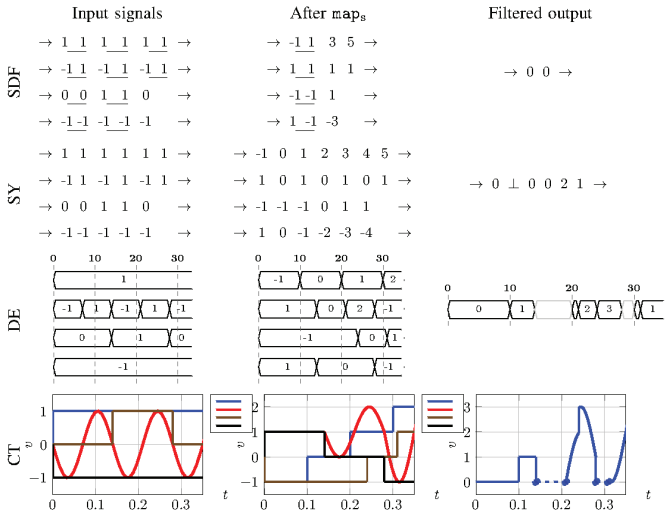
[‡] $\Gamma_{\text{SDF}}$ = (consumption rate for first input[, consumption rate for second input]), production rate

The projections onto the behavior layer in Fig. 2d and the function layer Fig. 2e grant us more insight about the structure of the system. Entities of higher layers are abstracted, e.g. the '△' process in 2d and 2e, or the '⊠' behavior in 2e. Since $T$ for timed MoCs is a strict order and processes are continuous, we can deduce that for MoCs like SY, DE, CT, projections in function layer, behavior layer and process layer are isomorphic. This sets up a rich common set of tools enabling design transformations, e.g. the displacement of synchronization points or the coalescing of functions. On the other hand, isomorphism between the behavior and process layer breaks for MoCs like SDF, where function arguments do not necessarily mirror input signals, requiring a separate but equally rich set of tools for analysis [23]. Fig. 2f depicts a possible implementation model using platform primitives, here a memory-based massively parallel multi-processor system.

Table II shows the response of `sys` (11) for signals of different MoCs. The SDF example underlines tokens consumed at the same time. Starting with the SY example, we can observe the effect of the $\texttt{when}_p$ process. The DE example is the distributed version of a HDL simulation and illustrates the effects of an explicit tag system. The CT example shows the manipulations on signals carrying functions over time.

We have implemented most principles presented in this paper as a shallow-EDSL in the functional programming language Haskell, and this example was designed and simulated as part of the suite [19].

TABLE II: SIMULATION OF THE SYSTEM IN EQ. (11)




## V. RELATED WORK

*Embedded system and CPS design* assumes an explicit or implicit computational model dictating the "laws of physics" that govern the interaction of components in a design. From ForSyDe [5] we have borrowed the concepts of *process constructor* as higher-order function and *structured signal*, modeled as infinite list. ForSyDe exploits the Bird-Merteens heritage for semantic-preserving and even nonsemantic-preserving transformations. While concerned only on the MoC aspects, ForSyDe provides a library of monolithic process constructors, while our framework aims at deconstructing and minimizing the set of building blocks. Ptolemy II [24] offers a plethora of building blocks whose execution is orchestrated by a *director* with respect to MoC semantics. In our opinion, the concept of director inhibits the exploitation of true concurrency among processes and constrains to express hierarchy within a model as locally homogeneous, whereas we advocate a pure functional model on all layers. As a first-order language, Ptolemy II models static structures and dynamism is expressed

ad-hoc, whereas we have demonstrated the power of higher order functions in combination with partial application to express complex structures and behaviors. For tackling the different notions of time from a functional perspective UNITI [6] describes multi-domain systems and a transformational environment based on *local solvers* for conversions between time domains and towards more efficient implementations. Functional Reactive Programming (FRP) [4] is a family of DSLs (Yampa, Fran, Frop, FAL, Fvision) for designing *hybrid systems*, i.e. with both continuous and discrete components, and enforces *transformers* as sole means to modify the transfer function of a process. Both UNITI and FRP use applicative functors for defining structured composition, but they specialize on their domain and lack a broader view on aspects such as behavior or untimed execution. DSLs for designing, simulating and *synthesizing* digital circuits, such as C$\lambda$ash [8] or Kansas Lava [7] demonstrate the applicability of the functional principles, but are restricted to a synchronous MoC.

*Parallel computing* flourishes with frameworks that exploit algorithmic skeletons to formulate patterns of parallel computation and communication. Only the functional approaches are discussed. Repa [13], is an EDSL which aims at describing transparent parallelism using a *purely-functional array interface* based on *parameterised, collective operations*, i.e. maps, folds, and permutations. Accelerate [14] and Nikola [15] push these ideas forward by providing dynamic code generation flows towards GPUs. Obsidian [16] tackles the separation of concerns between computation and orchestration by describing two sub-languages: a language of arrays and a language of GPU programs. While previous approaches exploit only array computations, Eden [17] is a general-purpose parallel functional language, with a comprehensive skeleton-based methodology, suitable for developing sophisticated skeletons, as well as for exploiting more irregular parallelism that cannot easily be captured by a predefined skeleton. None of the presented parallel languages treats timing nor behavioral aspects.

## VI. CONCLUSIONS

We have presented a formal framework rooted in functional programming for the modeling and analysis of CPS. We have bound concepts from different communities to describe an elegant and minimalist environment for modeling complex behaviors through hierarchical composition of *atoms*. As such, we aimed at pushing orthogonalization of concerns to its limits by defining clearly separated *layers* for abstracting different aspects in CPS, and describing intuitive means of interaction between them. We have shown the potential for analysis and synthesis by modeling a system which manifests concurrency and inherent parallelism and drawing its projection on different layers. Lastly, we have demonstrated the independence of the composition of atoms from the semantics carried by them, by injecting the example system with signals belonging to four different MoCs, and outputting the expected results.

For future work we plan: 1) the definition of a standalone DSL with a non-strict parameterized type system; 2) the modeling and exploitation of different types of structural parallelism and associated algorithmic skeletons, as pioneered by Eden; 3) the description of correct-by-construction flows based on *design transformations* and *design space exploration* as exhibited by ForSyDe.


## ACKNOWLEDGMENT

This work has been partially supported by the EU integrated project CONTREX (FP7-611146).


## REFERENCES


[1] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: orthogonalization of concerns and platform-based design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 12, pp. 1523–1543, 2000.

[2] E. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 12, pp. 1217–1229, Dec. 1998.

[3] H. J. Reekie, "Realtime signal processing," Ph.D. dissertation, School of Electrical Engineering, 1995.

[4] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, "Arrows, robots, and functional reactive programming," in *Lect. Notes Comput. Sc.* Springer, 2003, pp. 159–187.

[5] I. Sander and A. Jantsch, "System modeling and transformational design refinement in ForSyDe," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 1, pp. 17–32, January 2004.

[6] K. C. Rovers and J. Kuper, "UNITI: Unified composition and time for multi-domain model-based design," *Int. J. Parallel Prog.*, vol. 41, no. 2, pp. 261–304, 2013.

[7] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, "Introducing Kansas Lava," in *IFL*. Springer, 2009, pp. 18–35.

[8] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "CλaSH: structural descriptions of synchronous hardware using Haskell," in *2010 13th Euromicro Conference on DSD: Architectures, Methods and Tools*. IEEE, 2010, pp. 714–721.

[9] C. McBride and R. Paterson, "Applicative programming with effects," *J. Funct. Programming*, vol. 18, pp. 1–13, 2008.

[10] M. Cole, "Algorithmic skeletons," in *Research Directions in Parallel Functional Programming*. Springer, 1999, pp. 289–303.

[11] D. B. Skillicorn, *Foundations of parallel programming*. Cambridge University Press, 1995.

[12] J. Fischer, S. Gorlatch, and H. Bischof, "Foundations of data-parallel skeletons," in *Patterns and Skeletons for Parallel and Distributed Computing*, F. A. Rabhi and S. Gorlatch, Eds. Springer London, 2003, pp. 1–27.

[13] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier, "Regular, shape-polymorphic, parallel arrays in Haskell," in *ACM Sigplan Notices*, vol. 45, no. 9. ACM, 2010, pp. 261–272.

[14] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, "Accelerating haskell array codes with multicore GPUs," in *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, ser. DAMP '11. New York, NY, USA: ACM, 2011, pp. 3–14.

[15] G. Mainland and G. Morrisett, "Nikola: embedding compiled GPU functions in Haskell," *ACM Sigplan Notices*, vol. 45, pp. 67–78, 2010.

[16] J. Svensson, M. Sheeran, and K. Claessen, "Obsidian: A domain specific embedded language for parallel programming of graphics processors," in *Symp. Implement. Appl. Funct. Lang.* Springer, 2008, pp. 156–173.

[17] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí, "Parallel functional programming in Eden," *J. Funct. Programming*, vol. 15, no. 03, pp. 431–475, 2005.

[18] M. Jaskelioff and O. Rypacek, "An investigation of the laws of traversals," *arXiv:1202.2919*, 2012.

[19] "The ForSyDe-Atom library," https://github.com/forsyde/forsyde-atom, accessed: 2016-11-30.

[20] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[21] R. S. Bird, *An introduction to the theory of lists*. Springer, 1987.

[22] R. Bird and O. de Moor, *Algebra of Programming*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

[23] S. Stuijk, M. Geilen, and T. Basten, "SDF3: SDF for free." in *ACSD*, vol. 6, 2006, pp. 276–278.

[24] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity – the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.