ForSyDe-Atom: Taming Complexity in Cyber Physical System Design with Layers

GEORGE UNGUREANU, KTH Royal Institute of Technology, Sweden

JOSÉ EDIL GUIMARÃES DE MEDEIROS, University of Brasília, Brazil

TIMMY SUNDSTRÖM, INGEMAR SÖDERQUIST, and ANDERS ÅHLANDER,

Saab AB, Sweden

INGO SANDER, KTH Royal Institute of Technology, Sweden

We present ForSyDe-Atom, a formal framework intended as an entry point for disciplined design of complex cyber-physical systems. This framework provides a set of rules for combining several domain-specific languages as structured, enclosing *layers* to *orthogonalize* the many aspects of system behavior, yet study their interaction *in tandem*. We define four layers: one for capturing timed interactions in heterogeneous systems, one for structured parallelism, one for modeling uncertainty, and one for describing component properties. This framework enables a systematic exploitation of design properties in a design flow by facilitating the stepwise *projection* of certain layers of interest, the isolated analysis and *refinement* on projections, and the seamless *reconstruction* of a system model by virtue of orthogonalization. We demonstrate the capabilities of this approach by providing a compact yet expressive model of an active electronically scanned array antenna and signal processing chain, simulate it, validate its conformity with the design specifications, refine it, synthesize a sub-system to VHDL and sequential code, and co-simulate the generated artifacts.

CCS Concepts: • **Software and its engineering** → *System modeling languages*; • **Computer systems organization** → *Embedded systems*; • **Theory of computation** → *Models of computation*;

Additional Key Words and Phrases: Cyber-physical systems, system design language, models of computation, design methodology, modeling, simulation, validation, synthesis

ACM Reference format:

George Ungureanu, José Edil Guimarães De Medeiros, Timmy Sundström, Ingemar Söderquist, Anders Åhländer, and Ingo Sander. 2020. ForSyDe-Atom: Taming Complexity in Cyber Physical System Design with Layers. *ACM Trans. Embed. Comput. Syst.* 20, 2, Article 10 (December 2020), 27 pages.

<https://doi.org/10.1145/3424667>

This research was partially funded by the Swedish Governmental Agency for Innovation Systems, NFFP7 project # 2017-04892: Correct by construction design methodology (CORRECT).

Authors' addresses: G. Ungureanu and I. Sander, KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science, Division of Electronics and Embedded Systems, Kistagången 16, Kista, Sweden, SE-164 40; emails: {ugeorge, ingo}@kth.se; J. E. G. de Medeiros, University of Brasília, Campus Univ. Darcy Ribeiro, Ed. Faculdade de Tecnologia, Departamento de Engenharia Elétrica, Brasília, DF, Brazil, 70910-900; email: j.edil@ene.unb.br; T. Sundström and I. Söderquist, Saab AB, Business Area Aeronautics, Bröderna Ugglas Gata, SE-581 88, Linköping, Sweden; emails: {timmy.sundstrom1, ingemar.soderquist}@saabgroup.com; A. Åhländer, Saab AB, Business Area Surveillance, SE-412 89, Gothenburg, Sweden; email: anders.ahlander@saabgroup.com.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2020 Copyright held by the owner/author(s).

1539-9087/2020/12-ART10

<https://doi.org/10.1145/3424667>

1 INTRODUCTION

Cyber-physical system (CPS) design is a relatively recent engineering branch that combines computers and physical systems. Despite its success to deliver systems for industrial, medical, automotive, avionics, and many other application domains, CPS design is well known to be a vibrant research area, brimming with challenges due to the manifold functional and non-functional properties of systems. As a discipline, CPS design tries to combine different design approaches from distinct mature areas with established, but often incompatible, abstractions [28], lacking a common agreement on how to describe and interpret the different notions of *system behavior*. The active electronically scanned array antenna (AESA) radar [44] is a typical case of a high-complexity and high-performance component that needs to be integrated within a CPS. An AESA radar is a versatile system capable of extracting spatial and temporal information of incoming objects and consists of hundreds, or thousands, of antenna elements. Its processing chains need to compute extremely large quantities of data ranging from billions to trillions of floating-point operations per second. Especially for airborne applications, there are tight requirements on size, weight, and power (SWaP), together with robustness and safety [3, 4]. Although such safety-critical CPS are becoming more predominant in today's society, current design flows have difficulties to cope with the complexity and performance requirements for this kind of systems, and thus they are out of reach for commercial or public appliances.

The main tool humans have been using to tackle complexity is *abstraction*, i.e., the capturing of physical phenomena into pure, mathematical, analyzable *models*. Engineering is predominantly guided by the need to create: As engineers, we strive to build artifacts that faithfully mirror a chosen model, this in turn becoming the *specification* for how the artifacts should behave. This practice of materializing models into real objects has been possible by virtue of *model formalization* that enabled the natural stacking of models upon other models with a clear and unambiguous effect. For example, as of today we seldomly question whether a particular line of code in a computer program is in fact sloshing the right electrons in its processor's semiconductor to produce the desired result, even though there are several levels of abstractions and associated hazards between the two paradigms [29]. However, this degree of assurance starts to fade as we move to higher abstractions such as system level, typical to CPS. The multidisciplinary nature of the involved models makes it very challenging to adopt paradigms that fit together naturally, especially when considering rigorous, systematic design flows [42].

Another pillar of system design, as early pointed out by Keutzer et al. [26], is *orthogonalization* or *separation of concerns*, which means the decoupling of the different aspects of system design, such as computation, control, timing, or other perspectives of a system implementation, to tackle them independently. While promising a convenient frame to systematically solve design problems in a divide-and-conquer manner, orthogonalization is still largely limited due to the inherent interdependency of many of these aspects in CPS that requires them to be studied in tandem. Also the limited inter-operability of electronic design automation (EDA) languages and tools make it hard to combine the results of different analysis and synthesis methods, hindering the potential for finding optimal implementations or, even worse, increasing the risk of unwanted interactions found late in the development process [34]. In turn, this might cause additional development costs, delays, and serious system vulnerabilities.

Finally, any practical system design flow needs to be computer-aided; hence, to be understood and manipulated by computers, models need to be specified as formal languages. Comprised from a set of symbols and combination rules (*syntax*), and a set of rules for interpreting them (*semantics*), a language is *formal* if its syntax and semantics are based on well-formed, usually algebraic rules. A formalism gives value to the language from the perspective of analysis or design flow [17].

Following Sifakis' [42] guidelines for system design languages we advocate for (1) *minimality*, or at least economy of description; (2) *mathematical rigor*, or at least lack of ambiguity; and (3) *executability* to enable computer-aided verification or at least simulation. In the context of CPS, however, these requirements are conflicting with the need for expressing complex, multifaceted behaviors. Capturing too many aspects of CPS design within one language would render it non-scalable and unusable. Defining many "smaller" languages would cripple their interoperability and maintenance. A similar crux has been observed in the 1960s [27]: Hundreds of computing domains spawned thousands of ultimately obsolete domain-specific languages (DSLs).

1.1 The Key Idea

We present ForSyDe-Atom [1], an open-source, formal framework for modeling of CPS based on the functional programming (FP) paradigm, as an effort to tackle the challenges above, guided by three overarching principles:

- (1) we strive to push orthogonalization [26] to its limits, by "physically" (i.e., syntactically) separating the different aspects of CPS and capturing them in a model of structured, enclosing, interacting embedded DSLs, which we name *layers*;
- (2) we seek to exploit the principle of minimality [42], by deconstructing these DSLs to their basic semantics captured through primitive, composable building blocks called *atoms*;
- (3) we handle complexity in a compositional, hierarchical manner [34], by describing common building blocks in CPS as meaningful *patterns* of atoms.

1.1.1 Why Functional Programming? Addressing the language explosion in the mid-1960s, Landin [27] noticed that we rarely need to design new languages; much rather, we need to design *vocabularies* and host them within a well-defined, consistent, harmonious framework provided by a typed lambda calculus, i.e., a functional language. This idea is enforced a decade later when, in his Turing Award lecture, Backus [6] presented the functional style as being powered by an algebra of "combining forms hierarchically," which can use high-level programs to create still higher-level ones, where variables are programs themselves, i.e., *functions* or *abstractions*. Backus argues that this paradigm liberates programming from the von Neumann style dominated by the notion of *transitions* between *states*, which, inherited as a modeling paradigm, would deteriorate the description of any aspect that transcends computing machines. Hence we aim to abstract *behaviors* using pure functions.

1.1.2 How to Capture Orthogonal, Yet Interacting Aspects? A central construct in FP is the higher-order function (HOF), which is a function that takes functions as arguments and returns a function as result. The mechanism enabling layers is called *function lifting* [9], in which the callee is evaluated within the context of the caller and affects the semantics of the overall program. Yet, both the caller and the callee are unaware of each other and can be developed in isolation, whereas the underlying base of FP, lambda calculus enables them to interact in a deterministic way [18], granting both *orthogonalization* and *executability*. Driven by the case study, we exemplify four orthogonal layers interacting by virtue of the lifting principle, yet we do not claim that these are exhaustive for CPS design. On the contrary, through didactic examples we strive to inspire the reader to extend or even define own layers to cover for design aspects we did not consider in our case study.

1.1.3 Why Atoms? Coined by Jakobsen [23], atoms have originally been used as generic *aplicative* [32] operators to define *process constructors* [40]. While process constructors focus solely on capturing the semantics of models of computation (MoCs) [24], our *atom* is a generalization of its original concept: An atom (*el.* "indivisible") is a constructor capturing a primitive behavior,

supported by a formalism. In this sense, MoC theory, while remaining a central focus of our study, is just one of the formalisms enabling the description of system behaviors. Enhanced by the concept of layers, atoms describe a *generic* framework for combining other formal models alongside MoCs. Our aim with atoms is to describe small, maintainable, ideally minimal grammars for each DSL.

1.1.4 Why, Altogether? ForSyDe-Atom extends the Formal System Design (ForSyDe) modeling framework [40] and, as shown in Section 2, it is meant to be an entry point to formal and rigorous system design methodologies. The main idea is to provide designers with enough degrees of freedom to express different interacting aspects of CPS, but at the same time restrict them to using formal, minimalistic DSLs that can be designed, analyzed, and manipulated both in isolation and in tandem. While currently manifested, for proof-of-concept, as a framework shallow-embedded in the FP language Haskell, the ideas are general enough to fit a range of other potential implementations.

1.2 Contributions

- (1) We present the first comprehensive overview of the ForSyDe-Atom framework for modeling CPS, clarifying the main modeling concepts: layers, atoms, and patterns (introduced in Reference [46]), and introducing a new concept that enables systematic design flow: projection.
- (2) We detail these modeling concepts by defining four separate layers capturing different aspects of CPS modeling: timing (introduced in References [45, 46]), structured parallelism (introduced in References [46, 48]), uncertainty, and component properties.
- (3) We demonstrate the modeling framework through a formal, high-level, executable model of an AESA radar system composed of the signal processing chain (introduced in Reference [48]) alongside the radar environment dynamics.
- (4) We demonstrate the usefulness of such a framework as entry point to a systematic design flow by highlighting critical design steps on our case study (extending from Reference [48]): simulation and tracing, property formulation and validation, refinement and synthesis to hardware or software components, and co-simulation of the generated artifacts.

Section 2 introduces and defines, in a didactic manner, the three novel modeling concepts w.r.t. the key idea in Section 1.1, positioning them against the classical ForSyDe framework but also highlighting the potential use for orthogonalization. Driven by the case study, Sections 3–6 exemplify four separate layers w.r.t. the definitions in Section 2, further documented in Reference [1]. Founded on different formalisms, these layers and serving different purposes in system design, as shown in Section 7 that presents an overview of an industrial case study, extensively documented using literate programming in Reference [47].

2 MANAGING COMPLEXITY WITH ATOMS AND LAYERS

Models in EDA should play a dual role. On the one hand, models aid in designing systems and formulating specifications for what *correct behavior* means. In a design flow, however, models aid in achieving *the right implementation* that conforms with the said specification by means of successive design decisions and refinements that transform more abstract models into more concrete ones. Nonetheless, choosing the appropriate model for the right purpose (phenomenon, object, etc.) is by far a non-trivial task and is an engineering merit [29]. ForSyDe-Atom aims at providing an environment for combining and experimenting with multiple model domains yet restricting them to follow a set of rules to be useful within a disciplined design process. As such, models

are expressed in terms of interacting, or as we called them *layered*, DSLs. This section is a gentle introduction to the main concepts enabling this framework.

DSLs in ForSyDe-Atom are driven by data, or more precisely, by the way data are constructed. To describe data domains we use structured types [36].

Definition 1 (Structured Data Types). These are types built using *type constructors*, i.e., functions of the form $\alpha \rightarrow C(\alpha)$. These type constructors encode certain properties around an (arbitrary) base type α , describing a *structure* $C(\alpha)$ that enables certain *model behavior* when evaluated.

Example 1.1 (Streams). Let a language Str define the type of *stream* $S(\alpha)$ as a collection of elements *ordered in time*. Streams are defined recursively in terms of the precedence operator “<”

$$S(\alpha) = \emptyset \mid \alpha < S(\alpha),$$

which means “a stream of α is either nothing, or an element of α preceding another stream of α .” We use the notation $a : \alpha$ to qualify the type of a to be α . For example, the stream $1 < 2 < \emptyset : S(\text{Int})$ is interpreted as “element 1 : Int followed by element 2 : Int followed by nothing.”

The structure inferred from type constructors grants the machine-understandable alphabet that describes certain aspects of a system. *Behaviors*, and by that we refer to a system’s reactions to its input stimuli, can be modeled as functions on these structured types. Here, we employ the notion of HOF [9] to orthogonalize behaviors that are affected by the alphabet of the enclosing structure from the behaviors affected by the enclosed type. Before exemplifying HOFs, we introduce *partial application*, a mechanism based on currying [9] inherited from the host paradigm, which allows us to systematically apply functions one argument at a time, yielding intermediate functions with fewer arguments. The associated curried notation will be used throughout this article.

Example 1.2 (Curried Notation). In the example below, using curried notation one can represent function $f : \alpha \times \beta \rightarrow \gamma$ in an equivalent form $f : \alpha \rightarrow \beta \rightarrow \gamma$, which suggests that by binding an argument by partial application, e.g., a to 1, we obtain another function of the remaining arguments, e.g., $f \ 1 = g$,

$$\begin{array}{lcl} f : \alpha \times \beta \rightarrow \gamma & \equiv & f : \alpha \rightarrow \beta \rightarrow \gamma \\ f(a, b) = a + b & \equiv & f \ a \ b = a + b \end{array} \quad \begin{array}{l} \xrightarrow{a=1} \\ g : \beta \rightarrow \gamma \\ g \ b = (f \ 1) \ b = 1 + b. \end{array}$$

Example 1.3 (Higher-order Functions: map). Let a HOF in the language Str with the signature

$$\text{map}_{\text{Str}} : (\alpha \rightarrow \beta) \rightarrow S(\alpha) \rightarrow S(\beta),$$

defined in such a way that it takes a function $\alpha \rightarrow \beta$ and applies it on every element of $S(\alpha)$ yielding $S(\beta)$, a function $(+1) : \text{Int} \rightarrow \text{Int}$, which increments an integer with 1, and the stream $s = 1 < 2 < 3 < \emptyset : S(\text{Int})$. Then $\text{map}_{\text{Str}}(+1) \ s = 2 < 3 < 4 < \emptyset : S(\text{Int})$. Alternatively, we associate an infix operator $\bullet$ to map_{Str} to write more compact expressions $(+1) \bullet s = 2 < 3 < 4 < \emptyset$.

Definition 2 (Atom). An atom is an indivisible, composable, pure, i.e., side-effect free, operation on a specific structured type holding meaningful semantics.

Atoms enforce our commitment to deterministic, i.e., pure behavior models [29], as the goal of the framework is to provide implementation flexibility, where the relationship between the defined system inputs and accepted outputs is unaffected by implementation or operation choices [18].

Example 2.1 (Atoms on Streams). Considering the language of simple streams, where transformations are modeled as dataflow processes, map_{Str} is in fact a *process constructor*, as originally defined by Reekie [39]. However, map_{Str} is also a primitive behavior, whose role is to *lift* the semantics of a unary function on α into the domain of Str ; hence, as far as Definition 2 is concerned,

it is an atom. By itself, map_{Str} operates only on one stream. To combine multiple streams, we need to capture another behavior determined by a formal notion of metric. Let then the binary atom

$$\text{merge}_{\text{Str}} : S(\alpha \rightarrow \beta) \rightarrow S(\alpha) \rightarrow S(\beta)$$

defined w.r.t. the synchronous hypothesis [8] in such a way that it takes a stream of functions $S(\alpha \rightarrow \beta)$ and applies them pairwise on the elements of another stream $S(\alpha)$ yielding a stream $S(\beta)$. For example, if $s_f = (+1) < (-2) < (\times 3) < \emptyset : S(\text{Int} \rightarrow \text{Int})$, and $s_v = 1 < 2 < 3 < \emptyset : S(\text{Int})$, and if we associate an infix operator $(*)$ to $(\text{merge}_{\text{Str}})$ for brevity, then $s_f * s_v = 2 < 0 < 9 < \emptyset : S(\text{Int})$.

The functional host allows forming *any* expression in this DSL to describe combinational operations on streams of arbitrary elements by means of function composition and function application, as long as the type signatures match. However, there are two problems with this approach: (1) Not *all* compositions are legal w.r.t. a DSL's formalism and (2) using atoms alone for the purpose of design tends to get tedious and superfluous. This is why we introduce patterns as means to hierarchically capture complex behaviors as *libraries* of useful *design components*:

Definition 3 (Patterns). Patterns are meaningful compositions of atoms forming building blocks commonly used in CPS design.

Example 3.1 (The comb Pattern). Let the pattern defined in terms of $\text{map}_{\text{Str}} (\bullet)$ and $\text{merge}_{\text{Str}} (*)$

denote a generic combinational component on streams that takes an arbitrary function with n arguments and applies it on n merged input streams. For space reasons, we omit defining patterns with multiple outputs, but we point to Reference [46] for more information. The applicative mechanism [32], i.e., systematic partial application and lifting, is depicted above. If $f = (+) : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$, and the streams $s_1 = 1 < 2 < 3 < \emptyset : S(\text{Int})$ and $s_2 = 10 < 20 < 30 < \emptyset : S(\text{Int})$, then $(+) \bullet s_1 = (+1) < (+2) < (+3) < \emptyset : S(\text{Int} \rightarrow \text{Int})$ and thus $(+) \bullet s_1 * s_2 = \text{comb}_{\text{Str}}(+)_1 s_1 s_2 = 11 < 22 < 33 < \emptyset : S(\text{Int})$.

The reader familiar with the classic ForSyDe will recognize comb_{Str} as being equivalent to the `zipWithSY/combSY` class of process constructors. However, Example 3.1 defines it as a *generic* composition rule of formal primitive behaviors rather than a set of monolithic blocks. This enables a more disciplined and pragmatic language design.

Definition 4 (Layer). A layer is a DSL for describing an aspect, or viewpoint of CPS design and consists of the following: (1) one or more structured types with well-defined constructors; (2) a finite set of atoms with clearly defined semantics; (3) among these, one or more higher-order functions that lift pure functions into instances within its DSL, acting as conduit between layers; and (4) a set of legal and/or meaningful composition rules, enforced by a library of patterns for increased design expressiveness.

Example 4.1 (Layers in Tandem). Let there be (1) layer `Str` as presented in Examples 1.1–3.1 a language for representing behaviors on streams, (2) a language `Vec` for representing operations on vectors, and (3) a language of simple integer arithmetic operations. For now we are only concerned

Fig. 1. Vectors of streams design: sys_1 . Fig. 2. Stream of vectors design: sys_2 . Fig. 3. Examples of projections for component sys_1 in Figure 1.

that vectors are structured types $V(\alpha)$, where $\langle e_1, e_2, \dots \rangle : V(\alpha)$ captures the spatial distribution of elements $e_1, e_2, \dots \in \alpha$. Let the higher-order atom $\text{map}_{\text{Vec}} : (\alpha \rightarrow \beta) \rightarrow V(\alpha) \rightarrow V(\beta)$ defined in such a way that it maps an arbitrary function $\alpha \rightarrow \beta$ on every element of a vector. Two meaningful, yet *distinct* designs created using these DSLs are $sys_1 = \text{map}_{\text{Vec}}(\text{map}_{\text{Str}}(+1)) : (V(S(\alpha)) \rightarrow V(S(\alpha)))$ in Figure 1, which is a parallel network of stream processes, each process concurrently applying function $(+1)$ on its input stream; and $sys_2 = \text{map}_{\text{Str}}(\text{map}_{\text{Vec}}(+1)) : (S(V(\alpha)) \rightarrow S(V(\alpha)))$ in Figure 2, which is one process where each token in the stream is a vector of numbers, and its function can *potentially* be applied in parallel w.r.t. its data. Although using the same constituent elements, the two distinct *intersection of behaviors*, each modeled in its own layer, describe different temporal and spatial interactions.

The classical ForSyDe framework [41] has mainly been concerned in orthogonalizing function from temporal (MoC) behaviors via process constructors, hence an insightful reader might find some ideas later in Section 3 familiar. ForSyDe-Atom pushes this concept further and generalizes it, making possible to define other layers of formalisms as DSLs. The intersection of behaviors resulted from combining expressions in these DSLs is obtained, like in Example 4.1, by means of composition involving *lifting of behaviors*, hence the need for rule 3 in Def. 4. As a functional operation, lifting (e.g., functor [9]) does not distort the structure of an expression in a layer, but rather *embeds* the functionality coming from a layer below. This grants a rigorous means not only to construct, but also to deconstruct multi-layered models to constituent components:

Definition 5 (Projection). If a (sub-)system C is described in terms of layers $L_1, \dots, L_N$, then we say that its projection upon a layer $L_{i \in 1..N}$ denoted $\text{Pr}[L_i](C)$ is an operation that yields *only* the expressions described in L_i ; all information inferred from any layer above $L_{j>i}$ is discarded; all expressions from a layer below $L_{j<i}$ figure only as placeholders. For brevity, we denote $\text{Pr}[L_i, \dots, L_j](C) = \text{Pr}[L_i](C) \cap \dots \cap \text{Pr}[L_j](C)$ where $i > \dots > j$, i.e., the ordered re-combination (in our case by means of function composition) of the expressions resulted from projections.

Example 5.1. Consider sys_1 from Figure 1, re-drawn in Figure 3(a). This system can be projected upon different layers like in the following examples: Figure 3(b) reflects the parallel process farm, where the function applied on streams is left as a placeholder; similarly, Figure 3(c) shows a farm pattern, without consideration for what the placeholder f might be. In Figure 3(d), the contents from that placeholder are projected and isolated.

A more ample and formal discussion on layers and projections is left outside the scope of this article. We only point out that projections grant the possibility to select exactly which layers are of interest, giving only an enclosed reflection without affecting the semantics of the whole model. For example, an analysis step can make use of projections to extract sub-problems that can be solved in a divide-and-conquer manner; for model transformations, it provides an environment to refine-and-replace [33] not only by isolating a component (i.e., a sub-expression within a layer) but also specific behaviors of a component (i.e., across layers); for synthesis, it enables the separation of implementable parts from behavioral descriptions used for simulation and validation only.

Having as incentive the potential of orthogonalization in system design, we proceed to describe in Sections 3–6 four distinct layers used in CPS modeling in terms of Definition 4. Through these we indicate that the ForSyDe-Atom framework establishes an ontology and a set of rules to discipline the design of DSLs; however, the choice and design of the DSLs themselves are arbitrary.

3 MASTERING TIMED BEHAVIOR

One of the current challenges on CPS design is the nondeterministic nature of models that arise from the composition of otherwise deterministic models caused by lack of common temporal semantics [28]. We define a *MoC layer* founded on the *tagged signal model* [30] in which systems are regarded as *compositions* of *processes* acting on *signals*. Signals are defined as sets of events, $s = \{e_0, e_1, \dots\} \in S$, where each event $e_i = (t_i, v_i)$ has a tag and a value. A *tag system* establishes causal ordering between events and can model time, precedence, synchronization points, and so on. Based on how tag systems are defined, one can identify several *MoCs* as classes of behaviors dictating the semantics of execution and concurrency in a network of processes.

A process P is a set of possible behaviors, i.e., powerset, of signals or as a relation between multiple signals, i.e., intersection of powersets [30]. A process is *functional* if there exists a single-valued mapping between the input and the output signals that describes it, i.e., $F : S^{|I|} \rightarrow S^{|O|}$, where I and O determine the externally imposed behaviors over which P is defined, and thus a functional process is completely determined by the triple $P = (F, I, O)$. An important property of functional processes is that they are *determinate*. In engineering, we value determinate behaviors as specification models, because they are capable of defining how a correct implementation is supposed to act, offering a “golden model” toward which designs need to converge. As practical interpretation for determinate processes, Reekie [39] defined them as being themselves functional mappings over the (history of) input signals, capturing I and O in their type signature. Within this setting, he introduced *process constructors* as HOFs specifying generic rules of causal interaction between streams used to instantiate processes. This concept has been extended to encapsulate different MoC semantics [24] and has been a driving force behind the classical ForSyDe frameworks [41].

Our approach to defining a MoC layer is to embed a tagged signal model into a language of streams, thus inheriting the property of *causal precedence* between events. While only naively introduced in Section 2, streams are important, well-founded [9, 39] data structures, and processes on streams have spawned important models of computation and concurrency, as early as Reference [25]. With respect to MoC behaviors, we have identified four main atoms on signals:

- $\text{lift}_M(\text{infix } \odot) : (\Gamma, \alpha \rightarrow \beta) \rightarrow S(\alpha) \rightarrow S(\beta)$ is a HOF that takes a function from a layer below, (possibly) along with a context Γ , and lifts it to express MoC behavior.
- $\text{sync}_M(\text{infix } \otimes) : S(\Gamma, \alpha \rightarrow \beta) \rightarrow S(\alpha) \rightarrow S(\beta)$ synchronizes two signals while performing partial application on the functions in the layer below accordingly.
- $\text{pre}_M(\text{infix } \otimes) : S(\alpha) \rightarrow S(\alpha) \rightarrow S(\alpha)$ ensures *prefix order* in signals, by extracting a prefix behavior from the left-hand operand (LHS) and prepending it to the right-hand one (RHS).

Table 1. MoC Layer Definitions

MoC	event type	atom definitions			
		$\text{lift}_M / \odot$	$\text{sync}_M / \oplus$	$\text{pre}_M / \otimes$	$\text{shift}_M / \oplus$
SY	$\mathcal{E}_{SY}(\alpha) = \alpha$	$f \odot (v < s_e) =$ $f \ v < f \odot s_e$	$(f < s_f) \oplus (v < s_v) =$ $f \ v < s_f \oplus s_e$	$(v_i < _) \otimes s =$ $v_i < s$	$_ \oplus s = s$
DE	$\mathcal{E}_{DE}(\alpha) =$ $\alpha_{@T}$	$f \odot (v_{@t} < s_e) =$ $(f \ v)_{@t} < f \odot s_e$	$(f_{@t_1} < s_f) \oplus (v_{@t_2} < s_v) =$ $\begin{cases} (f \ v)_{@t_1} < s_f \oplus s_v & \text{if } t_1 = t_2 \\ (f \ v)_{@t_1} < s_f \oplus (v_{@t_2} < s_v) & \text{if } t_1 < t_2 \\ (f \ v)_{@t_2} < (f_{@t_1} < s_f) \oplus s_v & \text{if } t_1 > t_2 \end{cases}$	$(v_{@_} < _) \otimes s =$ $f_{@0} < s$	$s_d \oplus (v_{@t} < s) =$ $v_{@t+t_d} < s_d \oplus s$
CT	$\mathcal{E}_{CT}(\alpha) =$ $(\tau \mapsto \alpha)_{@T}$	$f \odot (g_{@t} < s_g) =$ $(f \circ g)_{@t} < f \odot s_g$	$(f_{@t_1} < s_f) \oplus (g_{@t_2} < s_g) =$ $\begin{cases} (f \circ g)_{@t_1} < s_f \oplus s_g & \text{if } t_1 = t_2 \\ (f \circ g)_{@t_1} < s_f \oplus (g_{@t_2} < s_g) & \text{if } t_1 < t_2 \\ (f \circ g)_{@t_2} < (f_{@t_1} < s_f) \oplus s_g & \text{if } t_1 > t_2 \end{cases}$	$(f_{@_} < _) \otimes s =$ $f_{@0} < s$	$s_d \oplus (g_{@t} < s) =$ $(t \mapsto g(t - t_d))_{@t+t_d} < s_d \oplus s$
SDF	$\mathcal{E}_{SDF}(\alpha) = \alpha$	$(\Gamma, f) \odot s_v =$ $f \ (\text{take } \Gamma \ s_v) <$ $f \odot (\text{drop } \Gamma \ s_v)$	$(\Gamma, f < s_f) \oplus s_v =$ $f \ (\text{take } \Gamma \ s_v) < s_f \oplus (\text{drop } \Gamma \ s_v)$	$(s_i < \emptyset) \otimes s =$ $s_i < s$	$_ \oplus s = s$

Fixity: func. appl. > atoms > precedence (<). Wildcard ($_$) = *don't care* value.

Fig. 4. Process constructors as patterns, instantiating.

shift_M (infix $\oplus$): $\mathcal{S}(\alpha) \rightarrow \mathcal{S}(\alpha) \rightarrow \mathcal{S}(\alpha)$ is the only means for manipulating tags in a way that ensures *strict causality*, i.e., increments all tags in the RHS signal with a strictly positive quantity in the given MoC's metric space, extracted from the LHS.

We claim that the four atoms of the MoC layer are sufficient to describe (strictly) causal, reactive behaviors in the condition that all input events are triggering computations as defined in Table 1. These atoms can build a wide range of process constructors as patterns, such as the ones seen in Figure 4. For space reasons an in-depth discussion about causality, ambiguity, deadlocks, a proof for this claim, as well as more complex behaviors is regarded as future work.

Definition 6 (MoC Layer). This layer is a DSL for describing timed interactions in *reactive concurrent systems*, founded on the theory of *tagged signals* [30]. With respect to Definition 4:

- (1) the enabling structured type is the *signal* $\mathcal{S}$, defined as stream S of events $\mathcal{E}$, i.e., $\mathcal{S}(\alpha) = \mathcal{S}(\mathcal{E}(\alpha))$, where the type definition of an event determines a *tag system*;
- (2) four atoms, lift_M , sync_M , pre_M , shift_M can describe complex reactive behaviors. These atoms capture different MoC semantics according to their input's tag system, as shown in Table 1.
- (3) lift_M is the HOF used as conduit between layers.
- (4) a set of composition rules needs to restrict *monotonic* and *causal* behaviors [30] as well as a wide variety of patterns enforce these rules, forming well-established *process constructors*.

Table 2. Example Simulation Traces for MoC Patterns (Figure 4)

Pattern	SY	DE	CT	SDF
comb _M (Fig. 4a)	$f a b c = a + b \times c$ $s_1 = 1 \quad 2 \quad 3 \quad 4 \quad 5$ $s_2 = 2 \quad 1 \quad 2 \quad 3$ $s_3 = 0 \quad 1 \quad -1 \quad 0 \quad 1$ $s'_1 = 0 \quad 3 \quad -5 \quad 0$	$f a b c = a + b \times c$	$f a b c = a + b \times c$	$f a b c = a + b_1 + b_2$ $s_1 = 1 \quad 2 \quad 3$ $s_2 = 0 \quad 0 \quad \dots$ $s'_1 = 1 \quad -1 \quad 2 \quad -2 \quad 2 \quad -2$ $s'_2 = 0 \quad -2 \quad -8 \quad -8$
moore _M (Fig. 4d)	$ms a = a - s$ $od s = s \times 2$ $s = 1 \quad 2 \quad 3$ $s' = 0 \quad \dots$ $s'' = 1 \quad 1 \quad 2$ $s''' = 0 \quad 2 \quad 2 \quad 4$	$ms a = a - s$ $od s = s \times 2$	$ms s(t) = 1 - s(t)$ $od s(t) = s(t) \times 2$	$\Gamma_{in} = \{c(1,1), p=2\}$ $ms s a = s_1 - a_1 < a_1 - s_1$ $\Gamma_{out} = \{c=2, p=1\}$ $od s = s_1 \times s_2 \times 2$ $s_1 = 1 \quad 2 \quad 3 \quad 4 \quad 5$ $s_2 = 2 \quad 1 \quad 2 \quad 3$ $s_3 = 0 \quad 1 \quad -1 \quad 0 \quad 1$ $s'_1 = 3 \quad 4 \quad 4 \quad 5$

Different MoCs dictate different execution and synchronization semantics. Based on the defined tag systems, several language sub-domains can be identified within the MoC layer. Although not a necessary trait, a pragmatic approach would be to overload atoms with a corresponding tag algebra, thus enabling to reuse the same atom patterns for all defined MoCs. A set of concrete examples of evaluating the same patterns according to different MoC semantics is shown in Table 2. We use the atom overloading approach to introduce four distinct MoCs used in the case study, pointing to Reference [1] for more information.

The Synchronous MoC. This is a well-established paradigm for modeling high-assurance safety-critical applications, modeled by languages such as Lustre, Esterel, and so on. [8]. In the synchronous (SY) MoC each signal in the system is synchronous with every other signal. Two events $e_1 = (t_1, v_1)$, $e_2 = (t_2, v_2)$ are synchronous, i.e., $e_1 \sim e_2$, if $t_1 = t_2$ and two signals are synchronous if $\forall e_1 \in s_1 \exists e_2 \in s_2$ such that $e_1 \sim e_2$ and vice versa. This implies two properties of SY systems: (1) any two events are either synchronous or unambiguously one precedes the other; (2) computation is assumed to be “instantaneous,” i.e., at the synchronization instants, data are always available.

For SY signals, we assume that each event’s tag is inferred from its position in the host stream, and thus the precedence operator ($<$) is enough to derive both a total order and a SY metric between signals. Since explicit tags are not necessary, we define the four atoms accordingly in Table 1: $lift_M$ recursively applies a function on all events on a signal, $sync_M$ performs recursive pairwise function application on two signals, pre_M assumes the first event from the LHS to be the prefix behavior of the RHS, whereas $shift_M$ propagates the RHS as it has no explicit tags on which to operate.

The Discrete Event MoC. This is a class of timed behaviors that concerns interactions defined at discrete instances in time, i.e., *timestamps*, such as the execution of digital or distributed systems. We define each event as an explicitly timestamped value of the form $\mathcal{E}_{DE}(\alpha) = \mathcal{T} \times \alpha$, and for any $e = (t, v)$ we use the notation $v_{@t}$. Assuming T an operation on signals that extract the tags from signal s , dropping the values, then $\mathcal{E}_{DE}$ is a proper DE tag system *iff*: (1) $\forall s \in \mathcal{S}(\alpha)$, $T(s)$ is a strictly incrementing sequence; (2) $\mathcal{T}$ is a set order-isomorphic to the set of integers, i.e., between any two tags t_u and t_v there is a *finite* number of possible tags [30]; and (3) $\mathcal{T}$ unambiguously defines a (practical) notion of synchrony, as requirement for representing time [29].

Following this interpretation, the discrete event (DE) atoms in Table 1 define an algebra over values and tags, among which: $sync_M$ performs the synchronization of events in timestamp order, conditioned through comparisons enabled by the third property above; pre_M ensures that the first event in a signal is always tagged with timestamp 0 considered global start, and hence our systems are one-sided [30]; $shift_M$ increments all tags in a signal with a positive t_d , which is extracted from s_d .

The Continuous Time MoC. This MoC is a class of timed behaviors that are defined over a *continuous span of time* but can also derive properties of causality and determinism between different

observations of the system. In engineering practices rather than talking about causality, continuous systems are described by their *dynamics*, i.e., how the system evolves over time. In Reference [45], we presented a means to represent continua by virtue of *abstraction*, and in Reference [16] we discuss a detailed algebraic approach to the continuous time (CT) MoC. As such CT signals are not being sampled and approximated but rather represented as pure functions from the start. This way, time and all its mechanical details such as numerical representation are being abstracted away from the algebra of atoms and become apparent only when an explicit evaluation is needed (e.g., to plot the signal).

Following Definition 6, $\mathcal{E}_{CT}(\alpha) = \mathcal{T} \times (\tau \rightarrow \alpha)$ is an extension of DE in the sense that the events carry functional mappings over the *complete span of time* $\tau \rightarrow \alpha$, where τ can be any numerical representation fit for simulation purposes. Much like DE, events are considered persistent, but now their discrete tag $\in \mathcal{T}$ represents timestamps when *discrete changes* in continuous functions occur. This interpretation is a practical representation of hybrid systems as discussed in Reference [28], but also enforces the view adopted in Reference [11], where continua are conservatively incorporated in programs under the supervision of discrete controllers. The CT atoms in Table 1 behave very much like their DE counterparts, except that lift_M and sync_M perform function composition ($\circ$) instead of function application, as well as shift_M modifies both the timestamp and the time domain of the sub-signal.

The Synchronous Dataflow MoC. This MoC belongs to the class of untimed behaviors, and describes applications with no data loss. They are particularly useful in describing streaming distributed applications as they do not over-specify the model with timing details. Formally, synchronous dataflow (SDF) is a special case of Kahn process networks (KPN) [25] where processes are also sequential with respect to their *firings* [30]. These firings derive partial ordering relations between events, hence describing *production* and *consumption rates*. In the case of SDF systems, the *sequence determinacy* restriction admits multiple time behaviors, the latter being regarded as an implementation refinement.

ForSyDe-Atom represents SDF systems by interpreting streams $\mathcal{S}$ simply as sequences $\Sigma(s)$ where Σ is an operation on signals s dropping their tags. Atoms take care of the partial order inferred from consumption c and production p rates, by passing these parameters as a context $\Gamma = \{c, p\}$ along with the lifted function. Let the stream functions take and drop select, and respectively discard, a number of events as dictated by Γ . Then, the SDF atoms in Table 1 act similarly to their SY counterparts, with the exception that lift_M and sync_M apply functions on sub-sequences according to Γ , as well as pre_M extracts the prefix from the entire LHS and prepends it to the RHS.

Interfaces. The vista of CPS lies in intertwining time semantics in models to describe the symbiosis between different kinds of processes. We have shown how to represent different tag systems using structured types, and in the case of the CT MoC representation, how to describe hybrid interactions by means of abstraction. Now we introduce the notion of *MoC interface* $\text{if} : \mathcal{S}_M \rightarrow \mathcal{S}_{M'}$ as a functional mapping from signals in MoC sub-domain M to signals in sub-domain M' .

From a pure *syntactic* perspective, translations are systematic re-wrappings of data within different containers with *no loss of information*. Let the bijective function $\text{trans}_{M \rightarrow M'} : \mathcal{E}_M \rightarrow \mathcal{E}_{M'}$ that takes an event from one MoC and projects its information according to another MoC. This function is unique and admits an inverse $\text{trans}_{M \rightarrow M'}^{-1} = \text{trans}_{M' \rightarrow M}$, ensuring that information is preserved and behaviors can be recreated across domains as “changes of perspective” rather than refinements. Syntactic translations can thus be expressed as stream mappings $\text{if} = \text{map}_{\text{str}} \text{trans}_{M \rightarrow M'}$ and examples are shown in Figure 5. Only *neighboring MoCs*, i.e., MoCs with similar time perspectives, benefit from natural translations between tag systems; according to Reference [5] interfacing between any two MoCs can be done by chaining neighbouring translation functions.

Fig. 5. Example of syntactic translation between different MoCs.

Fig. 6. A/D converter as comb_M pattern.

Fig. 7. Semantic equivalence of catamorphisms through promotion.

From a *semantic* perspective, though, interface designs need to answer several non-trivial questions such as How do domains perceive each other? What is an acceptable behavior of a realistic (i.e., lossy) domain interface? Can information loss be modeled in a deterministic, analyzable manner? How is causality preserved across time domains? While these questions could also be investigated in other modeling paradigms, e.g., contracts [7], some realistic interface systems can be described in terms of compositions of behaviors within the MoC layer. For example, a sample-and-hold A/D converter can be modeled like in Figure 6 as a specialized comb_M pattern between a DE trigger signal and a syntactically converted CT signal, by applying function $\text{sample_}(t \mapsto f\ t)_{@t_s} = t \mapsto f\ t_s$.

4 EXPLOITING PARALLELISM

The term *algorithmic skeletons* was coined by Cole [15] to denote high-level constructs encapsulating parallelism, communication, and synchronization, having an associated cost complexity. They originate from the need for programming languages designed from a fixed set of constructs capturing common computation patterns [6] and from the fact that many parallel applications share a common set of known interaction patterns like processor farms or pipelines [15, 20]. The key idea behind algorithmic skeletons is that there exists a set of functions on structured data types called *catamorphisms* [43] that respects the way in which objects of their type is built. If a and b are objects of a structured type and $\bowtie$ an operation that builds them into a larger structure, then h is a catamorphism *iff* there exists a binary operation $\otimes$ such that Equation (1) holds,

$$h(a \bowtie b) = h(a) \otimes h(b) \quad \text{CATAMORPHISM} \quad (1)$$

Intuitively, this shows that catamorphisms enable computation on separate parts of a large data structure (where each element is the carrier of an algebra) to be performed in parallel.

Skillicorn [43] identifies a broad spectrum of structures liable to catamorphisms called *categories*, such as sets, trees, arrays, and even graphs. For this article, we only use *vectors* (called *lists* in Reference [43]):

$$\mathcal{V}(\alpha) = \langle \rangle \mid \langle a \rangle \mid \langle a \rangle ++ \langle a \rangle,$$

defined in terms of three type constructors: $\langle \rangle : \mathcal{V}(\alpha)$, which instantiates an empty vector; $\langle a \rangle : \alpha \rightarrow \mathcal{V}(\alpha)$, which creates a singleton vector from element a ; and the infix $++ : \mathcal{V}(\alpha) \rightarrow \mathcal{V}(\alpha) \rightarrow \mathcal{V}(\alpha)$, which builds a vector from concatenating other two. $\mathcal{V}(\alpha)$ is a homogeneous structure *iff* $++$ is associative and the codomain for $\langle \rangle$ is the identity for $++$. Using these constructors we can build any vector, e.g., $\langle 1 \rangle ++ \langle 2 \rangle ++ \langle 3 \rangle$ is a vector of three numbers, conveniently abbreviated $\langle 1, 2, 3 \rangle$. With respect to Equation (1), $++$ plays the role of $\bowtie$ and infers a notion of *adjacency* or *locality*.

Table 3. Skeleton Layer Definitions

<p>$\text{farm}_S(\diamond) : (\alpha^n \rightarrow \beta) \rightarrow \mathcal{V}(\alpha)^n \rightarrow \mathcal{V}(\beta)$ $f \diamond (v_1, \dots, v_n) = f \diamond v_1 \diamond \dots \diamond v_n$</p>	<p>$\text{reduce}_S(\diamond) : (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \mathcal{V}(\alpha) \rightarrow \alpha$ $f \diamond v = f \diamond v$</p>	<p>$\text{pipe}_S(\diamond) : \mathcal{V}(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ $v_f \diamond a = ((\diamond \diamond v_f) a)$</p>	<p>$\text{recurs}_S(\diamond) : \mathcal{V}(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \mathcal{V}(\alpha)$ $v_f \diamond a = (\diamond a) \diamond (\text{tails}_S v_f)$</p>	<p>$\text{prefix}_S : (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \mathcal{V}(\alpha) \rightarrow \mathcal{V}(\alpha)$ $\text{prefix}_S f v = (f \diamond) \diamond (\text{tails}_S v)$</p>
<p>$\text{tails}_S : \mathcal{V}(\alpha) \rightarrow \mathcal{V}(\mathcal{V}(\alpha))$ $\text{tails}_S v = (\text{sel} \diamond) \diamond ((\diamond) \diamond v)$ where $\text{sel } x y = ((++ \text{first}) \diamond x) ++ y$ $\text{first} = (a \mapsto a) \diamond v$</p>	<p>$\text{take}_S : \mathbb{N} \rightarrow \mathcal{V}(\alpha) \rightarrow \mathcal{V}(\alpha)$ $\text{take}_S n v = (\text{sel} \diamond) \diamond ((\diamond) \diamond v)$ where $\text{sel } x y = \begin{cases} x + y & \text{if } \text{idx}_n(x) < n \\ y & \text{otherwise} \end{cases}$</p>	<p>$\text{get}_S : \mathbb{N} \rightarrow \mathcal{V}(\alpha) \rightarrow \alpha$ $\text{get}_S n v = \text{sel} \diamond v$ where $\text{sel } x y = \begin{cases} x & \text{if } \text{idx}_n(x) = n \\ y & \text{otherwise} \end{cases}$</p>	<p>$\text{groups}_S : \mathbb{N} \rightarrow \mathcal{V}(\alpha) \rightarrow \mathcal{V}(\mathcal{V}(\alpha))$ $\text{groups}_S n v = \text{take}_S n \diamond ((\text{drops } n, \dots) \diamond v)$</p>	<p>$\text{stencil}_S : \mathbb{N} \rightarrow \mathcal{V}(\alpha) \rightarrow \mathcal{V}(\mathcal{V}(\alpha))$ $\text{stencil}_S n v = \text{take}_S n \diamond (\text{take}_S (N_v - n + 1)(\text{tails}_S v))$</p>

Operator fixity: function application > atoms.

As such, for every vector $v : \mathcal{V}(\alpha)$ let $|v|$ denote the number of elements in v , and the function $\text{idx}_v(x)$ return the exact position of a particular element $x \in v$. Atoms in the skeleton layer are catamorphisms:

- $\text{map}_S(\text{infix } \diamond) : (\alpha \rightarrow \beta) \rightarrow \mathcal{V}(\alpha) \rightarrow \mathcal{V}(\beta)$ is a HOF that takes a function from a layer below and maps it to every element keeping the structure intact;
- $\text{merge}_S(\text{infix } \diamond) : \mathcal{V}(\alpha \rightarrow \beta) \rightarrow \mathcal{V}(\alpha) \rightarrow \mathcal{V}(\beta)$ merges two vectors element-wise while performing partial application on the functions “below” accordingly, and $|a \diamond b| = \min(|a|, |b|)$;
- $\text{reduce}_S(\text{infix } \diamond) : (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \mathcal{V}(\alpha) \rightarrow \alpha$ reduces a vector of elements to one element using a binary function.

An important theorem in the theory of algorithmic skeletons, *factorization* [20, 43], states that a function h is a catamorphism if it can be factorized like in Equation (2). As of this, the skeletons in Table 3, usually used in CPS design, are in fact patterns of map (i.e., the generic farm_S) and reduce (i.e., reduce_S). We classified them into *computation patterns*, which are HOFs capable of creating structures, i.e., networks, from entities in a layer below, respectively *communication patterns*, which are mainly used to permute or re-arrange elements in a vector for a specific purpose,

$$h = \text{reduce}(\otimes) \circ \text{map}(a \mapsto h \langle a \rangle), \quad \text{FACTORIZATION} \quad (2)$$

$$h \circ \text{reduce}(\bowtie) \equiv \text{reduce}(\otimes) \circ \text{map}(h). \quad \text{PROMOTION} \quad (3)$$

Another theorem, *promotion*, says that if $\otimes$ is associative, then Equation (3) holds. These two theorems enable a rich set of tools for reasoning isomorphic transformations toward more efficient implementations of elementary skeletons. In other words, systems described in terms of skeleton atoms can be derived in a design flow to fit a more appropriate implementation, especially when enhanced with a cost calculus [15]. As didactic example, let g an associative function and the derivation

$$\text{reduce}_S(g) \equiv \text{reduce}_S(g) \circ \text{reduce}_S(++) \circ \text{groups}_S \equiv \text{reduce}_S(g) \circ \text{farm}_S(\text{reduce}_S(g)) \circ \text{groups}_S,$$

where groups_S is used such that it splits the input vector in cost-balanced parts. This demonstrates how load-balancing a reduce_S catamorphism preserves the properties of its initial description.

Definition 7 (Skeleton Layer). This layer describes interactions in *structured parallel systems*, i.e., systems with inherent potential to be parallelized, and is founded on the theory of *algorithmic skeletons* [43]. With respect to Def. 4:

- (1) the enabling structured types are, in general, categories, and in particular, *vectors* $\mathcal{V}(\alpha)$;
- (2) as of Equation (2), three atoms, map_S , merge_S and reduce_S are sufficient to describe parallel behaviors.
- (3) map_S is the HOF used as conduit between layers.
- (4) meaningful compositions reasoned by Equation (2) enable the description of a wide variety of parallel patterns as well-established *catamorphisms*, e.g., see Table 3.

5 ASSESSING UNCERTAINTY

Models have a dual role in the engineering practice [28]: to explain physical phenomenon but also to assess physical systems w.r.t. a desired behavior. Modeling involves a choice about what to leave out of the system description and it is usual practice to leave uncertainty about our knowledge of the world out of it in initial design iterations. Random Variables (RVs) are the standard mathematical tool to bring the issue into consideration. They provide a deterministic framework to introduce environment uncertainty into the model, describing our imperfect knowledge and enabling assessment of how a specification will cope with it, and the degree and quality of variation the designer admits in a system implementation to capture a desired degree of robustness.

In probability theory [35], we say that a set S is the *certain event* when its elements represent possible outcomes of an experiment. The subsets of S are called *events*, the *impossible event* being the empty set. The *probability* of an event ξ is a number $P(\xi)$ such that (1) $P(\xi) \geq 0$, (2) $P(S) = 1$, and (3) if $\xi \cap \nu = \emptyset$ are mutually exclusive events, then $P(\xi \cup \nu) = P(\xi) + P(\nu)$. A RV x is a process of assigning an arbitrary number $x(\xi)$ to every outcome ξ of an experiment such that (1) the set $\{x \leq x\}$ is an event for every possible outcome of the experiment x and (2) the probabilities of the events $P\{x = \infty\} = 0$ and $P\{x = -\infty\} = 0$. Probability theory itself is not concerned with the meaning of the probabilities assigned to the potential values of x and the interpretation of probabilities is open to debate. A RV can be represented by a *probability density function* $f(x)$ if the integral $F(x) = \int_{-\infty}^x f(\zeta) d\zeta$, in which $F(x) = P\{x \leq x\}$, exists. In fact, given such a density function it is always possible to construct an experiment and an RV x with density $f(x)$.

We borrow the empirical definition of random numbers (RN) from Reference [35] given it is well suited to computer simulation: A sequence of numbers x_i is called random if its statistical properties are the same as the properties of random data obtained from a random experiment. If x_i are the samples of the RV x , then $y_i = g(x_i)$ are the samples of the RV $y = g(x)$. This suggests that, given an executable system model, sampling the input sequence from a RV would yield samples that characterize the output RV. However, we rely again in the notion of *abstraction* to represent RVs as *functions* and propagate them by means of function composition. This way the choice to sample a RV before feeding it to the system or at a later stage when plotting the outputs, is left to the designer.

In our context, a (pseudo) random number generator (RNG) is a pure function that takes a seed (represented as a *singleton type* $\in 1$) as input and returns a tuple with a RN and a new seed to be used in a subsequent call to the RNG. In this way, the RNG may be seen as an experiment that samples from S . We may then define the structured type to represent a RV distributed according to an arbitrary density function by wrapping a RNG such as a congruential generator on

$$\mathcal{D}(\alpha) = 1 \rightarrow \alpha.$$

This data structure abstracts away the machinery necessary to sequence subsequent calls to the RNG exposing only the “recipe” to get appropriate RNs on-demand. We define the atoms:

$\text{map}_N(\text{infix } \square) : (\alpha \rightarrow \beta) \rightarrow \mathcal{D}(\alpha) \rightarrow \mathcal{D}(\beta)$ is a HOF that takes a function f from a layer below and creates the RV $y = f(x)$ by mapping every point from the original RV according to f .

Fig. 8. Usage of probabilities as an own layer: RV definition; lifting of arbitrary function; sampling experiments.

$\text{trans}_N(\text{infix } \boxplus) : \mathcal{D}(\alpha \rightarrow \beta) \rightarrow \mathcal{D}(\alpha) \rightarrow \mathcal{D}(\beta)$ transforms one RV distribution to another by mapping samples.

$\text{sample}_N(\text{infix } \square) : 1 \rightarrow \mathcal{D}(\alpha) \rightarrow \alpha$ samples a RV by running one experiment trial defined by $\mathcal{D}(\alpha)$.

Definition 8 (Probability Layer). This layer is a DSL for describing uncertainty in values, and is supported by a *probability theory* [35]. With respect to Def. 4:

- (1) the enabling structures are recipes of type $\mathcal{D}(\alpha)$ as primitives for describing useful distributions, e.g., uniform, normal (Gaussian), exponential, and so on;
- (2) a set of three atoms, map_N , trans_N , and sample_N , introduce RVs into the layered framework.
- (3) map_N is the HOF used as conduit between layers.
- (4) combining recipes from multiple RVs can be expressed in terms of atom patterns, similarly to Section 2.

An example of summing three normally distributed RVs is shown in Figure 8 by lifting an arithmetic function into the domain of normal distributions. Figure 8(a) and 8(b) shows the potential values dictated by the corresponding recipes; yet to obtain an actual value, one needs to sample it. Figure 8(c) shows the accumulated results of 10^4 experimental runs. Please note the similarity of this layer with the CT MoC in Section 3: Both represent and propagate pure functions as data structures.

6 CAPTURING SYSTEM PROPERTIES

System design does not flow in a single direction from specifications to implementation but evolves by cross-checking and updating the two. Very often, we discover incomplete understanding in the specifications by implementing, and we solve unforeseen bugs by refining the specifications. While we advocate formal models to specify systems, we admit that in the majority of cases models are merely the estimations of incipient fuzzy ideas through the lens of design engineers. This brings up further questions: (1) How do we capture properties beyond what the modeling paradigm can describe? and (2) How do we avoid complete redesign if only parts of a system have been changed as a result of specification refinement? These challenges are tackled by the so-called *interface theories* [7] concerned with expressing (in)compatibilities between system components as relations on their inputs and outputs. The goal of these models is to provide frameworks for verification/validation of designs and refinements, as well as identification and formulation of design constraints.

One of the more pragmatic methods to ensure compliance of models with the design specifications is *property checking*, which consists in stressing a design under a set of arbitrary, legal, and meaningful inputs and observing whether its outputs satisfy a certain property. QuickCheck [13] is an existing framework for property checking that, although being successful mainly in software development [22], fits perfectly with our view of layered DSLs for CPS modeling. Its driving data type is a structure called *generator*, $\mathcal{G}(\alpha)$, which describes any potential (arbitrary) instance $a : \alpha$,

under the condition that α itself is a structured type with clear algebraic rules. Having “access” to the algebra of α , a generator is able to derive “smart” strategies for good coverage of random tests, e.g., cover edge cases first. A *property* then is a testable condition $\alpha \rightarrow \text{Bool}$, which is evaluated for every random test. An important side note is that property checking is incapable of testing outside the scope of the lifted DSL, i.e., properties need to be computable with respect to the (unambiguous) semantics defined in layer(s) below. For example, we cannot test temporal properties if a DSL below does not specify time. Associated with generators, we define following atoms:

alter_P (infix $\triangle$) : $(\alpha \rightarrow \beta) \rightarrow \mathcal{G}(\alpha) \rightarrow \mathcal{G}(\beta)$ alters a generator based on a lifted function;
 with_P (infix $\triangleleft$) : $\mathcal{G}(\alpha \rightarrow \beta) \rightarrow \mathcal{G}(\alpha) \rightarrow \mathcal{G}(\beta)$ combines two generators to create a third one;
 when_P (infix $\triangleleft$) : $\mathcal{G}(\alpha) \rightarrow (\alpha \rightarrow \text{Bool}) \rightarrow \mathcal{G}(\alpha)$ filters out unfeasible values from a generator;
 imply_P (infix $\Rightarrow$) : $\mathcal{G}(\alpha) \rightarrow (\alpha \rightarrow \text{Bool}) \rightarrow \text{Bool}$ creates a property on a generator.

Although a wide range of property expressions can be formed with these atoms, the syntax would be counter-intuitive, to say the least. Let then the following *syntactic sugar* rules:

$$\begin{aligned} \mathcal{G}(\alpha) &\equiv \forall a : \alpha & f \triangle \mathcal{G}(\alpha) &\equiv \forall x : \{f \ a | a : \alpha\} & \mathcal{G}(\alpha) \triangleleft p &\equiv \forall a : \alpha, p \ a \\ f \triangle \mathcal{G}(\alpha) \triangleleft \mathcal{G}(\beta) \triangleleft \dots &\equiv \forall x : \{f \ a \ b | a : \alpha, b : \beta, \dots\}. \end{aligned}$$

For example, the following property tests if the magnitude of complex numbers is within a predefined range

$$(a \ b \mapsto a + ib) \triangle (\mathcal{G}(\text{Float}) \triangleleft (a \mapsto a \geq -1 \wedge a < 1)) \triangleleft \mathcal{G}(\text{Float}) \Rightarrow (x \mapsto mg \ x \geq 0 \wedge mg \ x < \sqrt{2}),$$

where the function mg calculates the magnitude of a complex number, which can be re-written as

$$\forall x : \{a + ib \mid a : \text{Float}, a \in [-1, 1), b : \text{Float}\} \Rightarrow mg \ x \in [0, \sqrt{2}).$$

Due to the random nature of QuickCheck tests, the suites are capable of finding failing sequences of inputs, which are then *shrunk* to define minimum violating cases [13], making it very useful in development iterations. For example, testing the previous property using the QuickCheck engine would throw an error along the lines of `*** Failed! Falsifiable (after 3 tests): 0.9303092 :+ (-1.086389)`. Indeed, the property is incorrectly formulated, since the inputs to b are not properly delimited. Adding the clause $b \in [-1, 1)$ renders the suite successful, returning a message along the lines of `+++ OK, passed 100 tests`.

Definition 9 (Property Layer). This layer aids in describing and testing denotational properties on system models, enabled *exclusively* by the algebras of the layers below. With respect to Def. 4:

- (1) the enabling structured type is the *generator* $\mathcal{G}(\alpha)$;
- (2) four atoms, alter_P , with_P , when_P and imply_P can describe property test suites;
- (3) alter_P , when_P , and imply_P are HOFs used as conduits between layers;
- (4) *syntactic sugar rules* are used to hide atom expressions and imply specific patterns.

The property layer shows that the modeling principles of ForSyDe-Atom are general enough to apply to a wide range of not only new but also existing DSLs. As an example, we have embedded QuickCheck [13], a mature DSL for property checking, into our framework to make use of it as part of a layered design flow. We did that by ensuring consistency with Def. 4.

7 CASE STUDY: AN AESA RADAR SYSTEM MODEL

As case study to illustrate the potential of ForSyDe-Atom in a design flow, we consider the design of an AESA radar [44] system. We capture the main properties of such a system in a high-level model with three overarching goals: (1) *to understand* how it reacts to physical stimuli, (2) *to validate* that

Fig. 9. The AESA radar system specification.

the design meets the original textual specification, and (3) *to create* the system by highlighting a few strategies for rigorous refinement of the high-level model. Given these goals, Section 7.1 briefly introduces the *given* textual specification of the AESA system. Section 7.2 presents the system model of the environment, the data acquisition, and the signal processing chain using three layers (Section 3–5) and traces its behavior. In Section 7.3, we isolate model components, wrap them within a Property layer (Section 6), and validate that they are indeed modeling their specifications and no human errors have been introduced. Finally, Section 7.4 presents two potential transformational design flows toward digital hardware (e.g., FPGA), respectively software (e.g., embedded code) for *one* model component, by gradually (1) *projecting* its behaviors of interest according to Def. 5, (2) *refining* the projection using formal tools in the given layers’ paradigm, (3) *validating* the refined behaviors by property-checking them against the original specification, and (4) *replacing* the original component with the refined one. The refined component is synthesized mainly as syntactic translation, while the artifact generation and co-simulation is aided by existing tools in the ForSyDe ecosystem, e.g., ForSyDe-Deep [2]. A technical report, the full open source code and experiments for this article are found in Reference [47].

7.1 Specification

An AESA antenna, see Figure 9(a), may consist of thousands of antenna elements. The relative phases of the different elements can be set to create a constructive interference in the chosen main lobe bearing, so that the pointing direction can be set without any moving parts by virtue of a digital beam forming principle [47]. The signal processing operations are mainly composed of FIR filters, FFTs, matrix multiplications, and other similar calculations aiming to identify the distance from an object as well as its speed and traveling direction. This lends the application to highly parallel streaming implementations that can be distributed on spatial and temporal dimensions; hence, our interest is to capture mainly spatial (using skeletons) and temporal properties (using MoCs).

The AESA signal processing chain is sketched in Figure 9(b) and assumes one stream per antenna element. The indata are organized into a sequence of “*video cubes*,” each corresponding to a certain integration interval. Each sample in the cube represents a particular antenna element, pulse, and range bin. Samples arrive with a constant rate defined by the physics of the antenna.

The main “knobs” for dimensioning the algorithms are controlled by four constants: N_A is the number of antenna elements in the radar, N_B is the number of beams formed to capture the different areas of interest to be evaluated, N_{bin} is the number of range bins accumulating data after each radar pulse, and N_{FFT} is the number of pulses necessary to perform a Doppler filter bank. In this article we consider an AESA system with 16 antenna elements, as shown in Figure 9(c), where $N_A = 16$, $N_B = 8$, $N_{bin} = 1,024$, and $N_{FFT} = 256$. As our skeleton-based model is parametric, scaling up the application resumes to changing these constants and providing the appropriate coefficients.

Fig. 12. DBF proc. net.

(a) As MAV catamorphism

(b) As systolic proc. net.

Fig. 13. FIR operation representations. Layers: ■ same model; ■ changed model.

represent that by wrapping the obtained complex numbers into an $\mathcal{D}$ type that defines a normal distributed RV by applying the Box-Muller transformation [12] over a uniform RNG and pass them downstream as functions [47, sec. 4.2]. Whenever we need a concrete value, we need to sample_N it. For model brevity, we apply sampling immediately, hence degenerating the probability layer. If the design requires so, then it may be propagated throughout the system.

7.2.3 Digital Beamforming. $DBF : S_{Mat}(M_{SY}(C))$ ¹ This step creates a number of simultaneous receiver beams, or “listening directions” from the input data. This is done by doing weighted dot product between the vector of antenna samples from a sampling period and a $N_A \times N_B$ matrix of beamforming coefficients, so that constructive interference is created in the desired bearings of the beams. Considering N_A synchronous (SY) signals as input, the DBF component applies a matrix-vector dot product. However, the basic operations are *not* arithmetic functions, but rather lifted SY interactions modeled with comb_S processes, hence yielding N_B output *signals* [47, sec. 6.1.2.1], as shown in Figure 12. This is another good depiction of the versatility of layer orthogonalization.

7.2.4 Pulse Compression. $PC : S_{Vec}(M_{SDF}(S_{Vec}(S_{Vec}(C))))$ The goal of this stage is to collect all received energy from one target into a single range bin. The received echo of the modulated pulse is passed through a matched filter, performed as a moving average (MAV) over all range bins gathered during the same pulse for each beam. We model this operation as a SDF process [47, sec. 6.1.2.2] applying the fir catamorphism in Equation (4) and shown in Figure 13(a), over a vector of N_{bin} samples consumed every firing from each channel, where *tail* is a function that drops the first element in a vector, and *coef* is a vector of coefficients,

$$\begin{aligned} \text{fir } \text{coef} &= S.\text{farm11 } \text{applyMav} . S.\text{recur } S.\text{tail} & \text{firs}(\text{coef}, v) &= (\text{farm}_S \text{ avg} \circ \text{recur}_S \text{ tail}_S)(v) \\ \text{where } \text{applyMav} &= S.\text{reduce } (+) . S.\text{farm21 } (*) \text{ coef} & \text{where } \text{avg}(v) &= (\text{reduces}_S(+) \circ \text{farm}_S(\times) \text{ coef})(v) \end{aligned} \quad (4)$$

The actual ForSyDe-Atom implementation of the *fir* skeleton, where *farm_mn* denotes a *farm* applied on *m* vectors and yielding *n* vectors, is shown side by side with its formal definition. Being hosted on a functional programming language, the resemblance between the source code and the mathematical description is uncanny, and thus we stick to the latter, more compact notation.

7.2.5 Corner Turn. $CoT : S_{Vec}(M_{SDF}(S_{Mat}(C)))$ To be able to calculate the Doppler channels, a rearrangement of data must be performed between functions that process data in range and pulse directions, called a corner turn. At the highest level, this simply means that for each beam channel, the pulse-bin matrices are being transposed. Hence, our CoT process consumes $N_{FFT} \times N_{bin}$ tokens corresponding to this matrix; and it transposes it and produces $N_{bin} \times N_{FFT}$ tokens. To

¹We use notation $S_{Mat}(\dots) \equiv S_{Vec}(S_{Vec}(\dots))$ and $S_{Cube}(\dots) \equiv S_{Vec}(S_{Vec}(S_{Vec}(\dots)))$ for convenience whenever adequate.

maximize data reuse, given the indata video size is two times the outdata size (see Figure 9(c)), we split the datapath into two concurrent processing channels with 50% overlapped data [47, sec. 6.1.2.3]. These channels are merged back into a stream of *interleaved* (re-built) cubes just before the INT stage as seen in Figure 14.

7.2.6 Doppler Filter Bank. DFB : $S_{\text{Vec}}(M_{\text{SDF}}(S_{\text{Vec}}(\mathbb{C}, \mathbb{R})))$ This is a chain of stages applied on all pulses belonging to each range bin. It gives an estimation of the target's speed relative to the radar and an improved signal-to-noise ratio due to a coherent integration of the indata. The pulse bins in the dataset are transformed into Doppler channels, after which the envelope (i.e., real data) is extracted. The Doppler channels are computed using a n-point radix 2 decimation in frequency FFT algorithm [47, sec. 6.1.2.4], which is also described in terms of pure catamorphisms [43].

7.2.7 Constant False Alarm Ratio. CFAR : $S_{\text{Vec}}(M_{\text{SDF}}(S_{\text{Mat}}(\mathbb{R})))$ This stage is intended to keep the number of false targets at an acceptable level while maintaining the best possible sensitivity. It normalizes the video to maintain a constant false alarm rate where the the sensitivity will be adapted to the clutter situation in an area of interest. The defining equations are given in the report [47, sec. 2.2.2.5], which consist in applying a series of algorithms performed on 2D stencil_s sets of neighboring elements (see Table 3), for every bin-pulse matrix in each beam [47, sec. 6.1.2.4], justifying the rates in Figure 14.

7.2.8 Integrator. INT : $S_{\text{Vec}}(M_{\text{SY}}(S_{\text{Matrix}}(\mathbb{R})))$ This is an 8-tap unsigned FIR filter that integrates over complete, interleaved video cubes originating from both channels. As compared with the FIR performed during PC stage, where the window slides in space domain over vectors of numbers, INT does integration in time domain over SY streams of cubes. Consider the base catamorphism fir' , stripped off its core functions from Equation (5). Replacing g_{rec} , g_{frm} , and g_{red} in Equation (5) with Equations (6), we obtain through promotion (Equation (3)) exactly the fir in Equation (4) and Figure 13(a). On the other hand, if we replace these arguments with Equations (7), respectively, where $+$, $\times$, and 0 are lifted to cubes, then we get the systolic process network in Figure 13(b), used in the INT stage [47, sec. 4.1.2.5],

$$\begin{aligned} \text{fir}'_S \ g_{\text{rec}} \ g_{\text{frm}} \ g_{\text{red}} \ \text{coef} \ \text{input} \\ = (\text{reduces}_S \ g_{\text{red}} \circ \text{farms}_S \ g_{\text{frm}} \ \text{coef} \ (5) \quad \begin{cases} g_{\text{rec}} = \text{tails} \\ g_{\text{frm}} = \langle \rangle \circ \text{reduces}_S(+) \circ \text{farms}_S(\times) \text{coef} \\ g_{\text{red}} = ++ \end{cases} \quad (6) \quad \begin{cases} g_{\text{rec}} = \text{delay}_M(0 < \dots) \\ g_{\text{frm}} = c \mapsto \text{comb}_M(\times c) \\ g_{\text{red}} = \text{comb}_M(+) \end{cases} \quad (7) \\ \circ \text{recurs}_S \ g_{\text{rec}})(\text{input}) \end{aligned}$$

Expressing FIR as a process network respectively as a vector operation shows a glimpse of the true “plug-and-play” nature of lifting, enabling layered designs.

7.2.9 The AESA Processing Chain. As seen in Figure 14, we have chained the blocks in Sections 7.2.3–7.2.8 and, for modeling efficiency, we have *fused* [9] the outer skeleton layers for the blocks in Section 7.2.4–7.2.7 [47, sec. 6.1.3]. Testing it against the reflection signals for 12 known objects (Section 7.2.1–7.2.2) reveals that the model is sane. In Figure 15, we see a snapshot from the aligned input samples from one antenna element (Figure 15(a)) against the signal processing output in its respective beam (Figure 15(b)). The output shows 12 detected objects, whose value represents the signal power, and placement on the Y and X axes represents their speed and direction respectively the relative distance.

7.3 Properties and Validation

While an initial trace inspection (Figure 15) should render our model “acceptable”, a number of unforeseen factors might still jeopardize its feasibility. To minimize the risk of unwanted behaviors tracked late in the design process, we formulate and test a set of properties by lifting components to

Fig. 14. Video processing chain (below). Video data processing direction (above).

Fig. 15. Simulation.

Table 4. Component Properties. Aliases $S = \mathcal{S}(\alpha)$, $V = \mathcal{V}(\mathcal{S}(\alpha))$

#	Comp.	Pre-condition	$\Rightarrow$ Property	Section	Dep.	#tests	Passed
1	DBF	$\forall i : V, i > 0 \Rightarrow$	$ dbf(i) = N_B$	7.2.3		100	✓
2		$\forall o : \{dbf(i) i : V, \alpha = [-1 - i, 1 + i]\} \Rightarrow$	forall $e \in o$ then $e \in [-1 - i, 1 + i]$	7.2.3		200	✓
3	PC	$\forall i : V, i > 0 \Rightarrow$	$ pc(i) = i $	7.2.4		100	✓
4		$\forall o : \{pc(i), i : S, \alpha = [-1 - i, 1 + i]\} \Rightarrow$	forall $e \in o$ then $e \in [-1 - i, 1 + i]$	7.2.4	2	200	✓
5	PC'	$\forall i : S, i' : S, \Sigma(i) = \Sigma(i') \Rightarrow$	$\Sigma(pc(i)) = \Sigma(pc'(i'))$	7.4.1	4	80	✓
6	PC''	$\forall v : \{toQ19(v) v : \alpha, v \in [x80000 + x80000i, x7FFFF + x7FFFFi]\} \Rightarrow$	$fromQ19(v) \in [x80000 + x80000i, x7FFFF + x7FFFFi]$	7.4.1	5,4	100	✓
7	PC [†]	$\forall s : S \Rightarrow$	$pc''(s) = pc^\dagger(s) = pc^\ddagger(s)$	7.4.1	6,5,4	100	✓
8	PC*	$\forall s : S \Rightarrow$	$pc(s) = pc^*(s)$	7.4.2		100	✓
9	CoT	$\forall(r, l) : \{cot(s) s : S, s \text{ large enough}\} \Rightarrow$	foreach t then $r[t] = l[t + \frac{NFPT}{2}]$	7.2.5		100	✓
10	INT	$\forall coef : \mathcal{V}(\alpha), s : S, s = \text{impulse signal} \Rightarrow$	$int(coef)(s) = \text{impulse response}$	7.2.8		100	✓

We assume strict ($=$) over α .

the *property layer*. This aims to both formalize loose specifications, e.g., see the table in Figure 9(c), but also to validate (and isolate) that each design is in fact correct, e.g., coefficients are rightly scaled.

Table 4 defines some properties in terms of the syntax proposed in Section 6 as *pre-condition* $\Rightarrow$ *statement* where pre-conditions are defining generators for legal inputs under which the statement is evaluated, and the statement infers an executable program resulting in a truth expression. The “Comp.” entries suggest which component from Figure 14 is targeting a certain property; “Section” points to which section discusses that component; “Dep.” marks dependencies between properties; “#tests” shows the number of tests in one batch, generated from the same random seed. The notation $s[t] = e$ is used to select an event e with tag t from signal s .

Properties such as 1, 3, and 9 test the sizes of the vectors involved in skeleton operations, whereas properties like 2, 4, and 6 test the range and scaling of numbers so that they ultimately fit on the specified platforms. In property 9, we test that the data displacement and re-alignment is always preserved during the corner turn, whereas property 10 tests whether our FIR design is behaving properly as a filter under all conditions. More properties, their detailed explanation, as well as source code can be found in Reference [47]. After sufficient iterations and corrections of both the design and specifications, all tests passed every run, and we could proceed to the next design phases.

7.4 Refinement, Synthesis, and Co-Simulation

For the purpose of demonstration, we choose the high-level PC component (Section 7.2.4) and synthesize it targeting both FPGA platforms and embedded processors. We co-simulate the resulting

Fig. 16. Layer structure for a PC worker during refinements: ■ marks layers for projection-refinement-reconstruction.

artifacts together with the high-level ForSyDe model in Figure 14. The synthesis is performed as gradual projection-refinement-reconstruction cycles, as marked in Figure 16. We verify along the way that the properties in Table 4 hold. To simplify the case study, we assume that the potential for parallelism of the outer farm skeleton in PC is satisfied by having enough resources to map each worker in parallel, e.g., new entities in VHDL or parallel cores in C. Hence, for brevity, we study the case for a single PC worker.

7.4.1 To VHDL. For this purpose, refinement strategies need to consider that the target platform paradigm is in fact register transfer level (RTL), whose semantics is most fitting with the SY MoC rather than SDF. Transforming SDF to SY would require imposing a total order from an otherwise untimed behavior. Thankfully, this is rendered trivial by our knowledge of the system data acquisition: Each token arrives with a *global sampling rate* from each antenna's (sync'd) ADC.

R1. Untimed MAV to timed FIR. To exploit this knowledge on timing behavior, we describe PC in terms of a streaming systolic process network like in Figure 13(b), whose silicon cost is $O(|coef|)$ and throughput f_s , rather than a fully parallel MAV network like in Figure 13(a), whose cost is $O(|coef| \times N_{bin})$ and which would operate in bursts of N_b samples. By virtue of catamorphism promotion as shown in Section 7.2.8, we are able to refine PC into PC', where each pc' process is behaving according to the SY MoC. When changing the time domain, we lose the information on consumption, so we need to explicitly describe a mechanism to flush the FIR state after N_{bin} samples have been processed. One solution is to replace g_{rec} in Equation (7) with a reset-counter that flushes pipeline after N_{bin} cycles [47, sec. 7.1]. This refinement *does not* preserve the time semantics of PC; much rather, it preserves sequence-equivalence [30], tested with property 5 in Table 4, which is sufficient w.r.t. our initial specification. This refinement alters only MoC and skeleton layers.

R2. Floating point to Q19. Although floating point numbers are common in DSP applications their implementation in hardware is expensive. Based on the initial specs in Figure 9(c), a better number representation is fixed point Q19. The transformed PC' [47, sec. 7.2] alters only the function (i.e., innermost) layer of the PC' as to operate in Q19. To integrate PC' within the AESA system we wrap the worker process within $fromQ19 \simeq toQ19^{-1}$ like below. Although these conversions imply quantization errors, conformance with the intended behavior under legal inputs is enforced by property 6 in Table 4,

$$comb(fromQ19) \circ PC''(toQ19 \diamond coef_s) \circ comb(toQ19).$$

R3. Balancing the FIR reduction. As seen in Figure 13(b), the last stage of the fir' network consists of a reduction that, in the most general case, is a linear tree with depth $O(n)$. However, as this reduction is performing element-wise summation that is associative, according to the promotion theorem it can be done with $O(\log n)$ depth instead. The refined PC[†] [47, sec. 7.4] uses, instead of the generic $reduce_s$ skeleton, a semantically equivalent load-balanced $reduceLog_s$ as defined in Figure 7.

R4. Deep Language Embedding. A *natural fit* between two languages means, in loose terms, that both have equivalent semantic constructs. A natural fit enables a higher-level language to be embedded [42] in a lower-level one (ideally) only as syntactic substitutions. The previous refinements were necessary so that our layered model reaches a natural fit with the RTL paradigm, in particular with the VHDL language. To handle the clerical details of code generation, we use ForSyDe-Deep [2], a code synthesizer in the ForSyDe ecosystem. This tool defines VHDL templates for all the patterns used in our Skeleton and MoC layers, plus the contained functions. Translating $PC^\dagger$ to $PC^\ddagger$ becomes simply choosing equivalent building blocks, whereas co-simulation is supported by using the same host language and data representations. Semantic equivalence between $PC^\ddagger$, $PC^\dagger$, and PC'' (and hence sequence equivalence with PC) is tested through property 7 in Table 4.

S1. FPGA Synthesis and Co-simulation. In this phase, we generate VHDL code for the PC component [47, sec. 7.4.2]. This is automated by ForSyDe-Deep, which creates: (1) a set of synthesizable VHDL files and libraries for the ForSyDe components, (2) a ForSyDe-to-VHDL simulation wrapper, and (3) a project structure to be imported by Intel Quartus tool suite. Using the ModelSim simulation wrapper, we test that property 7 and its parent properties hold. Simulating the new AESA system against the test input data detects the exact same objects, e.g., Figure 15, albeit at slightly different power values.

7.4.2 To Sequential C. Starting from *the same* layered model of the PC worker process, we consider mapping it on a embedded processor running imperative C code. In this case, the synthesis strategies are contrasting from the previous one and need to account for the fact that to reach a natural fit, the concurrent nature of MoCs, skeletons, and pure functions needs to be degenerated to a *sequence of actions*. Due to their untimed characteristics, SDF models have been long studied as entry points to software-oriented design flows. For space reasons, we only mention that refining a SDF model involves (1) calculating a sequential schedule, trivial in our case, and (2) taking care of the necessary *glue mechanisms* for synchronization, data acquisition, and production. Leaving it as an exercise to the reader, we focus instead on the synthesis of the inner fir_s skeleton, under the assumption that whenever it is executed, it has access to N_{bin} new samples in the local memory.

R1. Promotion and Fusion. Consider the FIR expressed as a MAV operation in terms of catamorphisms like in Figure 13(a). Flattening and translating mot-à-mot each of its skeletons into equivalent sequential programs would result in many inefficient *for*-loops over intermediate arrays.

Our scope is to isolate, and systematically transform, the given catamorphism into an equivalent forms via equational reasoning [43], for which each component has a corresponding (more efficient) software design pattern. The equivalence in Equation (8) has been demonstrated in Section 7.2.4 using promotion. Equation (9) is left as an exercise (see Table 3).

Let the fusion in Equation (10) describe a $shiftFarm_s$ pattern that represents array computations with incremental indexing where each iteration operates on the array remainder. Also, let the fusion in Equation (11) represent chained computations on intermediate arrays into a single *for*-iteration. Then $fir_s \equiv fir'' = shiftFarm_s(farmReduce_s(\times)(+))$,

$$reduce_s(++) \circ farm(\langle \rangle) \circ reduce_s(+) \circ farm_s(\times)coef \equiv farm_s(reduce_s(+) \circ farm_s(\times)coef), \quad (8)$$

$$recur_s\ tail_s \equiv tail_s, \quad (9)$$

$$farm_s(f) \circ tail_s \equiv shiftFarm_s(f), \quad (10)$$

$$reduce_s(g) \circ farm_s(f) \equiv farmReduce_s\ f\ g. \quad (11)$$

R2. Data Initialization. Due to its generality, implementing a vanilla $reduce_s$ implies initial condition testing that might be unnecessary in particular cases. Let a function g and id the

Fig. 17. Template selection and expansion.

```

1 void fir (complex_t* output, complex_t* input){
2   unsigned i, j;
3   for (i = 0; i < N_BIN; i++){
4     unsigned _range = N_BIN - i;
5     complex_t* acc = mkComplex(0,0);
6     for (j = 0; j < min(_range, N_COEF); j++) {
7       acc = cpxAdd(acc, cpxMul(input[i+j], COEF[j]));
8     }
9     output[i] = acc;
10  }
11 }

```

shiftFarm_s expanded in lines 3–10
 farmReduceI_s expanded in lines 5–9
 cpxMul and cpxAdd expanded in line 7

Listing 1. Synthesized FIR C code.

identity element for g . We define $\text{reduceI}_s g id = \text{reduce}_s g(\langle id \rangle ++ v) \equiv \text{reduce}_s g v$. This property equips designers with the means to exploit application knowledge into choosing a better-suited implementation: If id exists (in the FIR case case g is $+$ and id is 0), then reduction can be implemented with an initialized accumulator instead. Thus refining to PC^* implies replacing fir'' with $\text{fir}^* = \text{shiftFarm}_s(\text{farmReduceI}_s(\times)(+))$. We validate it with property 8 in Table 4.

S1. Template Selection, Code Generation, and Co-Simulation. The previous refinements were necessary to reach a *natural fit* with the target platform’s paradigm. While for the scope of our didactic example it is sufficient to qualify PC^* as “acceptable,” a design flow would need to employ a cost-enabled design space exploration taking platform aspects into consideration to reach the most appropriate form for an expression. Similarly to the VHDL example, we take care of code generation by substituting every pattern in our layered model with an equivalent code template in the target language, as per the principle of language embedding [42]. This process is visualized in Figure 17, where each placeholder (e.g., f or g) triggers the expansion of a new template to obtain the code in Listing 1. The information in black is extracted from the PC^* model; red parameters are inferred from the caller template. Appropriate target equivalents for functions on complex numbers are invoked, e.g., `mkComplex`, `cpxAdd`, and `cpxMul`. We test the synthesized C code along with the AESA high-level model by embedding fir^* into each PC process using a foreign function interface.

A final remark is that the same rationale of systematically expanding templates can be applied to any ForSyDe-Atom pattern expressed as a HOF as long as (1) there exists a (certified) target code template that implements it and (2) there exist equivalent data-type representations in the target language. Hence, the advantages of layered design for code synthesis are palpable.

8 RELATED WORK

We turn our attention to a few selected works from recent EDA research that aim at bridging the abstraction gap between *behavior specification* and *implementation* through formal means, focusing on their entry points. SCADE [14] amasses three decades of scientific and industrial experience and is associated with a methodology and a set of tools for modeling, validating, verifying, and deploying fully certified concurrent software for safety-critical applications such as avionics and automotive. Among the more influential languages associated with it we mention Lustre [8] for modeling SY reactive systems, Zélus [11] for continuous dynamics with differential equations, and Lutin [38] for modeling non-deterministic reactive systems. While inspiring some of our layers from these languages and adhering to their principles of correctness-by-construction, we are less concerned to *conservatively* extend our framework based on mature knowledge but rather to

offer the flexibility to breach into other research domains by first working with layered (formal) languages in tandem and then isolating problems for rigorous design using existing tools.

Architecture languages offer a higher layer of abstraction over existing languages to capture complex orchestration aspects. Prelude [21] adds real-time primitives to enable the programming of multi-periodic systems by assembling pure SY systems (e.g., in Lustre). Lingua Franca [31] leverages a semantic notion of time and event by wrapping a deterministic *reactor* model around native code (e.g., C, Java) to exploit real-time predictable platforms. The notion of layers in ForSyDe-Atom can be viewed as a binding framework as well, and we also show how to integrate existing languages and engines (e.g., QuickCheck) for the purpose of design activities.

The modeling, analysis, and synthesis of parallel applications has been an intensely studied subject. SkePU [19] is one of the approaches founded on the theory of algorithmic skeletons [15, 43], which proposes the use of skeletons as libraries of efficient code templates, similarly to our idea in Section 7.4.2. Array-OL [10] is another approach exploiting parallel patterns while deriving rigorous temporal behaviors by analyzing data dependencies. In one sense, it combines the theory of (synchronous) dataflow MoCs with parallel patterns, or, in terms of this article, it studies the skeleton and MoC layers in tandem, proposing valuable design methods and tools in this respect.

ReWire [37] is an eDSL with a semantics-driven design flow for designing, implementing, and verifying high-assurance, provably correct hardware. Its functional underlying principles make it conducive to formal verification. Sharing the inherent functional principles with ForSyDe, ReWire is a valid source of inspiration for future design practices in enforcing correctness.

9 CONCLUSIONS AND FUTURE WORK

We have presented a formal layered framework that captures functional, temporal, spatial, probabilistic, and relational properties of CPS in orthogonal, executable DSLs, interacting through the FP principles of abstraction and lifting. Using these DSLs, we have modeled, simulated, and validated the complete processing chain of an AESA antenna against a model of its environment. One block in the chain was refined toward hardware and software platforms, demonstrating how the framework supports synthesis, co-simulation, and validation.

While using existing formalisms associated with each layer, we have introduced a novel generic means to seamlessly combine multiple paradigms for the purpose of system design. The framework enables the systematic exploitation of formal models by allowing both the *construction* of complex behaviors as layers to be studied in tandem and the *projection* of individual behavioral aspects to be studied in isolation. Its practical potential has been indicated throughout the case study. The power of the design method is not limited by the layer concept but mainly by the formalisms supporting these layers. In turn, as future work we plan to exploit this concept by further developing or integrating new modeling paradigms as layers but also to apply previous experience in HW/SW codesign by using ForSyDe-Atom as part of a complete transformational system design flow.

REFERENCES

- [1] [n.d.]. ForSyDe-Atom API. Retrieved from <https://forsyde.github.io/forsyde-atom/api/>.
- [2] Alfonso Acosta. 2007. *Hardware Synthesis in ForSyDe*. Master's thesis. KTH Royal Institute of Technology.
- [3] ARP4754A. 2010. Guidelines for development of civil aircraft and systems. *SAE International* (2010).
- [4] ARP4761. 1996. Guidelines and methods for conducting the safety assessment process on civil airborne system and equipment. *SAE International* (1996).
- [5] Seyed-Hosein Attarzadeh-Niaki and Ingo Sander. 2016. An extensible modeling methodology for embedded and cyber-physical system design. *Simulation* 92, 8 (2016), 771–794.
- [6] John Backus. 1978. Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs. *Commun. ACM* 21, 8 (1978), 613–641.

- [7] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas A. Henzinger, Kim G. Larsen, et al. 2018. Contracts for system design. *Found. Trends Electr. Des. Autom.* 12, 2–3 (2018), 124–400.
- [8] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (2003), 64–83.
- [9] Graham Hutton. 2016. *Programming in Haskell* (2nd ed.). Cambridge University Press, USA.
- [10] Pierre Boulet. 2007. *Array-OL Revisited, Multidimensional Intensive Signal Processing Specification*. Ph.D. Dissertation.
- [11] Timothy Bourke and Marc Pouzet. 2013. Zélus: A synchronous language with ODEs. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*. 113–118.
- [12] G. E. P. Box and Mervin E. Muller. 1958. A note on the generation of random normal deviates. *Ann. Math. Statist.* 29, 2 (06 1958), 610–611.
- [13] Koen Claessen and John Hughes. 2011. Quickcheck: A lightweight tool for random testing of haskell programs. *ACM SIGPLAN* 46, 4 (2011), 53–64.
- [14] J. Colaço, B. Pagano, and M. Pouzet. 2017. SCADE 6: A formal language for embedded critical software development (invited paper). In *Proceedings of the 2017 International Symposium on Theoretical Aspects of Software Engineering (TASE’17)*. 1–11.
- [15] Murray I. Cole. 1989. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman London.
- [16] José E. G. de Medeiros, George Ungureanu, and Ingo Sander. 2018. An algebra for modeling continuous time systems. In *Proceedings of the Design Automation and Test in Europe Conference (DATE’18)*. 861–864.
- [17] Stephen Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. 1997. Design of embedded systems: Formal models, validation, and synthesis. *Proc. IEEE* 85, 3 (March 1997), 366–390.
- [18] Stephen A. Edwards. 2018. On determinism. In *Principles of Modeling*. Springer, 240–253.
- [19] August Ernstsson, Lu Li, and Christoph Kessler. 2018. SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *Int. J. Parallel Program.* 46, 1 (2018), 62–80.
- [20] Jörg Fischer, Sergei Gorlatch, and Holger Bischof. 2003. Foundations of data-parallel skeletons. In *Patterns and Skeletons for Parallel and Distributed Computing*, Fethi A. Rabhi and Sergei Gorlatch (Eds.). Springer, London.
- [21] Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. 2010. A real-time architecture design language for multi-rate embedded control systems. In *Proceedings of the 25th ACM Symposium on Applied Computing*. 527–534.
- [22] John Hughes. 2007. QuickCheck testing for fun and profit. In *Practical Aspects of Declarative Languages*, Michael Hanus (Ed.). Springer, Berlin, 1–32.
- [23] Mikkel Koefoed Jakobsen. 2013. *Formal Methods for Design and Simulation of Embedded Systems*. Ph.D. Dissertation. Technical University of Denmark.
- [24] Axel Jantsch. 2004. *Modeling Embedded Systems and SoC’s: Concurrency and Time in Models of Computation*. Morgan Kaufmann.
- [25] Gilles Kahn. 1974. The semantics of a simple language for parallel programming. In *Proceedings of the 6th International Federation for Information Processing Congress (IFIP’74)*. 471–475.
- [26] Kurt Keutzer, A. Richard Newton, Jan M. Rabaey, and Alberto Sangiovanni-Vincentelli. 2000. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 19, 12 (2000), 1523–1543.
- [27] Peter J. Landin. 1966. The next 700 programming languages. *Commun. ACM* 9, 3 (1966), 157–166.
- [28] Edward A. Lee. 2016. Fundamental limits of cyber-physical systems modeling. *ACM Trans. Cyber-Phys. Syst.* 1, 1, Article 3 (November 2016), 3:1–3:26 pages.
- [29] Edward A. Lee. 2018. Models of timed systems. In *Formal Modeling and Analysis of Timed Systems*, David N. Jansen and Pavithra Prabhakar (Eds.). Springer International Publishing, Cham, 17–33.
- [30] Edward A. Lee and Alberto Sangiovanni-Vincentelli. 1998. A framework for comparing models of computation. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 17, 12 (1998), 1217–1229.
- [31] Marten Lohstroh, Ínigo Íncar Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. 2019. Reactors: A deterministic model for composable reactive systems. In *Cyber Physical Systems. Model-Based Design*. Springer, 59–85.
- [32] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (2008), 1–13.
- [33] Seyed Hosein Attarzadeh Niaki and Ingo Sander. 2011. Semi-formal refinement of heterogeneous embedded systems by foreign model integration. In *Proceedings of the 2011 Forum on Specification & Design Language (FDL’11)*. IEEE, 1–8.
- [34] Pierluigi Nuzzo. 2019. From electronic design automation to cyber-physical system design automation. In *Proceedings of the 2019 International Symposium on Physical Design (ISPD’19)*.
- [35] Athanasios Papoulis and S. Unnikrishna Pillai. 2002. *Probability, Random Variables, and Stochastic Processes*. Tata McGraw-Hill Education.

- [36] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.
- [37] Adam Procter, William L. Harrison, Ian Graves, Michela Becchi, and Gerard Allwein. 2015. Semantics driven hardware design, implementation, and verification with ReWire. In *ACM SIGPLAN Notices*, Vol. 50. ACM.
- [38] Pascal Raymond, Yvan Roux, and Erwan Jahier. 2008. Lutin: A language for specifying and executing reactive scenarios. *EURASIP J. Embed. Syst.* 2008, 1 (2008), 1–11.
- [39] Hideki John Reekie. 1995. *Realtime Signal Processing—Dataflow, Visual, and Functional Programming*. Ph.D. Dissertation. University of Technology at Sydney.
- [40] Ingo Sander and Axel Jantsch. 2004. System modeling and transformational design refinement in forsyde. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 23, 1 (2004), 17–32.
- [41] Ingo Sander, Axel Jantsch, and Seyed-Hosein Attarzadeh-Niaki. 2017. ForSyDe: System design using a functional language and models of computation. *Handbook of Hardware/Software Codesign* (2017), 99–140.
- [42] Joseph Sifakis. 2013. Rigorous system design. *Found. Trends Electr. Des. Autom.* 6, 4 (2013), 293–362.
- [43] David B. Skillicorn. 2005. *Foundations of Parallel Programming*. Number 6. Cambridge University Press.
- [44] George W. Stimson. 2014. *Introduction to Airborne Radar* (3rd ed.). SciTech Publishing.
- [45] George Ungureanu, José E. G. de Medeiros, and Ingo Sander. 2018. Bridging discrete and continuous time models with atoms. In *Proceedings of the 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE'18)*. 277–280.
- [46] George Ungureanu and Ingo Sander. 2017. A layered formal framework for modeling of cyber-physical systems. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'17)*. IEEE, 1715–1720.
- [47] George Ungureanu, Timmy Sundström, Anders Åhländer, Ingo Sander, and Ingemar Söderquist. 2019. *Design of Sensor Signal Processing with ForSyDe: Modeling, Validation and Synthesis*, version 0.3.1. Technical Report. KTH Royal Institute of Tehnology. DOI: <https://doi.org/10.13140/RG.2.2.21573.81126>
- [48] George Ungureanu, Timmy Sundström, Anders Åhländer, Ingo Sander, and Ingemar Söderquist. 2019. Formal design, co-simulation and validation of a radar signal processing system. In *Proceedings of the 2019 Forum on Specification & Design Language (FDL)*. 1–8. <https://doi.org/10.1109/FDL.2019.8876905>

Received February 2020; revised July 2020; accepted September 2020