Int J Parallel Prog
DOI 10.1007/s10766-012-0226-5

UniTi: Unified Composition and Time for Multi-domain Model-based Design

Kenneth C. Rovers · Jan Kuper

Received: 5 April 2012 / Accepted: 5 October 2012
© Springer Science+Business Media New York 2012

Abstract To apply model-based design to embedded systems that interface with the physical world, including simulation and verification, current tools fall short. They must provide mathematical (model) definitions that stay close to the specification of the system. They must allow multiple domains, such as the continuous-time, discrete-time and dataflow domain, in a single model including well-defined interaction. They must support model transformations for refining a model during development. And most importantly, they must accurately include and simulate different notions of time in the model. UniTi is a model-based design flow and modelling and simulation environment that delivers on all these aspects. It is based on components that are signal transformations, and therefore mathematical functions. However, in each domain the representation of a signal differs. As components have the same structure in each domain, we can use unified composition operators to represent multiple domains in a single model. Furthermore, this composition provides a unified perspective on time in the domains, even though we differentiate between different notions of time. Time becomes a local property of the model, allowing us to represent and simulate time transformations such as time delays exactly without losing efficiency. Finally, model transformations are defined for such components, which are used for refining and developing the model and which are guided by the design steps in the design flow. We will formally define the domains, composition operators and transformations of UniTi and verify the approach with a case study on a phased array beamforming system.

K. C. Rovers (✉) · J. Kuper
Computer Architectures for Embedded Systems Group, Faculty of Electrical Engineering, Math and Computer Science, University of Twente, Enschede, The Netherlands
e-mail: K.C.Rovers@utwente.nl

J. Kuper
e-mail: J.Kuper@utwente.nl

Published online: 21 November 2012

Keywords Cyber-physical systems · Embedded systems · Model-based design · Multi-domain · Model transformations · Unified composition · Unified time

1 Introduction

For the design of future embedded systems it is essential that their interaction with the (physical) environment is an integral part of the design process. The environment captures the physical aspects required for correct operation of the computational aspects of the system. Such systems are also called cyber-physical systems.

A model-based design process is employed to effectively manage the complexity of designing and simulating these systems. However, as a consequence the model consists of multiple domains: the environment and analogue hardware are modelled in the continuous-time (CT) domain, and the discrete-time (DT) domain is used to model fixed digital hardware. Many embedded systems also contain programmable hardware in the form of (multiple) processors on a system-on-chip (SoC) connected by a networkon-chip (NoC). It is impractical to model the software running on such programmable hardware as a clock cycle accurate (fixed) digital hardware specification in the DT domain, as this contains too much detail and requires too much implementation effort for the first stages of the design process as well as causing simulation to become very slow. Therefore we use the dataflow (DF) domain to provide a higher level abstraction of the software running on the programmable hardware. The DF domain is very useful to represent parallel programs on multi-core architectures [21]; the DF model enforces explicit communication thereby providing the relevant dependencies for parallelisation, automatically takes care of synchronisation, and is analysable for deadlock, throughput and latency. However, there are only a few tools supporting the DF domain in mixed-domain models [1]. Moreover, in most tools the interaction between domains is not well-defined [3].

Model-based design provides a promise to allow iterative development by refining a single model from specification to implementation. At each refinement step, detail is added which is continuously simulated, verified and validated. Such refinement steps are ideally model transformations, which should be correctness-preserving. Unfortunately, in practice the model is typically manually developed and transformed by a designer and the only model transformation applied is for code generation. But code generation transforms part of the model to another language (e.g. VHDL or C), thereby losing the benefits of a single model. Model transformations within a language, for example for parallelizing an application for multi-core architectures, have very limited support in current state-of-the-art multi-domain modelling and simulation tools [3,7,30].

Of paramount importance in mixed CT and DT systems is a unified view of time across the domains. However, in digital hardware and software time is essentially abstracted away and it is therefore not straightforward to integrate the digital hardware and software domains with the CT domain [11]. More importantly, we have found no tool that distinguishes different notions of time. Different notions of time are necessary to provide an exact implementation of time transformations, such as a time delay, while retaining efficiency [21,23,24]. Current tools extract a set of

equations from the model and use a solver on these equations at a global time step for simulation. Therefore, these tools use a discrete global time step and the buffering of values at these steps to implement a time delay, making the simulation either inaccurate because of interpolation or inefficient because of using a very small time step.

UniTi is a design flow and modelling environment to solve the above problems. UniTi supports models in the CT, DT and DF domains and defines the interaction between domains. This includes a unified perspective on time and the distinction between different notions of time. A new execution model is defined for the DF domain that is based on the same semantics of components and signals as used in the CT and DT domains. UniTi is mathematical in nature because the specification of an embedded system is usually given in (or at least supported by) a mathematical form. Additionally, it allows for correctness preserving transformations and offers a unified abstraction mechanism to integrate the CT, DT, and DF domains. Finally, the problem with time transformations is solved in UniTi by using a local time and applying higher order functions or signal transformations that operate on functions of time. This gives us exact time delays and local solvers among others.

These claims are substantiated with a case study of a phased array with beamforming running on a multi-core architecture. This system is a good case study for UniTi as it involves multiple domains and is sufficiently complex to benefit from a model-based design approach, although we will use a simplified system for practical reasons. Beamforming systems are particularly problematic for mixed-domain simulation as many small continuous time delays are used to model the signals from the array.

The UniTi design flow and framework presented in [22] is extended in the present paper with the formalisms of the execution model for the DF domain, a detailed treatment of unified time, efficient simulation based on continuations, and a more detailed case study. The contributions of UniTi and this paper are to:

- identify and formally distinguish the domains (Sect. 3),
- provide unified composition of multi-domain components (Sect. 4),
- support different notions of time and unification over the domains enabling exact and efficient time transformations and local solvers (Sect. 5),
- provide efficient simulation using composition and continuations (Sect. 6),
- present a design flow based on model tranformations with UniTi (Sect. 7),
- illustrate UniTi with a case study (Sect. 8).

2 Related Work

An extensive survey of languages and tools for multi-domain systems can be found in [1]. Some of the better known tools are MATLAB/Simulink for CT and DT modelling and finite state models, Ptolemy (II) [3] which supports many domains, being a testbed for multi-domain modelling with a strong basis on DF, SystemC-AMS [30] as an extension of SystemC for system-level mixed signal modelling, and Modelica [7] which is a modelling language oriented towards bi-directional energy flow diagrams.

2.1 Mathematical Definitions

The model-based design approach typically uses a formal specification, which provides a mathematical description of the functionality of the system. The specification thus provides a convenient initial model, against which model transformations can be verified. As such, it is very useful to represent models using mathematical definitions.

Mathematical specification are not typical in modelling tools, contrary to intuition. For example, MATLAB, SystemC and Ptolemy II all use imperative languages, and Modelica uses an object-oriented declarative modelling language. UNITI includes a multi-domain modelling and simulation framework in a functional language. A functional language, being based on mathematical functions, is close to mathematics providing a nice fit to represent UNITI in. A key feature exploited is the use of higher-order functions, used for model interactions and model transformations.

ForSyDe [25,26] is also a system modelling and design refinement approach for embedded systems in a functional language. The first version of ForSyDe supports synchronous/reactive (SR) and discrete event (DE) models [25]. Inclusion of other domains such as the CT and DT domain is ongoing work.

The field of functional reactive programming (FRP) [4,9] uses higher-order functions, a standard feature of functional languages, to model the CT and time-ordered discrete events for the SR domain. FRP has made excellent progress in being applied to different domains (for example animation [4], user interfaces [2] or robotics [19]) and providing formal semantics. The original work [4] focused on interactive animation with switching behaviours (animation) and events (interaction). In later work explicit behaviours and events are combined and only signal transformations are used [9,31] in order to avoid space and time leaks (i.e. a backlog of remaining old data in the memory or large remaining computations that have not yet been evaluated). In [28] higher-order transformations are introduced as strategies. Strategies complement an algorithm with a parallelisation approach. Our framework applies the best aspects from these approaches to multi-domain model-based design.

2.2 Multi-domain Support

There are many tools supporting the CT and DT domains, with Simulink the de facto standard. They often implement DT signals as piece-wise CT signals [1]. However, there are few tools that support CT, DT *and* DF; we only know of Ptolemy and to some extent SystemC-AMS (which uses timed dataflow). We have found no formalisation or implementation of dataflow that matches the semantics of signals and components as used in CT and DT signal flow diagrams.

Together with UNITI and ForSyDe, Ptolemy and SystemC are also the only tools that do not use fixed domains [3], i.e. in these tools the supported domains can be extended with additional domains by specifying their interaction with the existing domains. Ptolemy uses an actor model and tagged signals for integration; signals are a collection of events and an event is a pair of a time and value [14]. Other tools also use time-value pairs. UNITI uses functions of time as signals, allowing exact time transformations and different notions of time (see below).

2.3 Model Transformation Support

The Object Management Group (OMG) provides a specification of the kind of models and diagrams used for model-based design as part of the SysML profile of UML [17], and the abstraction levels as part of the model-driven-architecture (MDA) specification. How model transformations are performed is still actively researched. Current support for model transformations is typically limited to code-generation [8,18].

Simulink and SystemC-AMS have no support for modelling transformations, except code generation to C or an HDL. Ptolemy only recently added initial support for model transformations using higher-order components [6], a similar but less flexible approach than used by UNITI (Sect. 7). Projects researching automated system level modelling with model transformations are Sesame [5] and Daedalus [16]. However, their focus is on automatically parallelising applications, not system modelling.

2.4 Exact Continuous Time Domain Support

All the researched tools and languages [1,7,30,33] use ordinary differential equation (ODE) solvers for simulating the CT domain; an equation system is set up and a global time step is applied for numerical approximation, thereby implementing CT signals as a sequence of values. Time transformations such as a time delay therefore buffer values and interpolate between available values introducing inaccuracies caused by the modelling tool [21,23,24].

Tools represent signals as either a value at a global time step or a time-value pair. Ptolemy has a notion of "super-dense time"; time is a real number with an index for ordering events that have the same time such as with discontinuities in a signal. Ptolemy and SystemC-AMS also offer some support for influencing the step size. For example, a block in a Ptolemy model can reject a step size of the ODE solver until all blocks agree. SystemC-AMS supports module and port time step propagation as a consistency check. Modelica is only a modelling language definition; the simulation engine is unspecified. There are several implementations of the language such as Dymola or MapleSim, yet to the best of our knowledge all tools use a global time step. Thus, all of the above tools do not offer possibilities to adequately support time.

FRP *does* use functions of time, but does not identify and use different notions of time; time is considered to be global and time is progressed globally by the FRP framework, depending on e.g. the processing load [4].

Besides a global time step, most tools also use a global solver, i.e. all equations are solved using the same time step. This is not very efficient as not all parts of the model require the same accuracy. Because of the use of a local time, UNITI uses and requires local solvers.

3 Formalisation of the Domains

In this section we propose a novel way to define and simulate models using the CT, DT and DF domains. In the CT domain we consider signals as *functions of time* such that the values of a signal can be exactly determined at every instant during the simulation.

In the DT domain we consider signals as values, which are considered piecewise horizontal in the CT domain. Thus, our simulation technique coincides with the standard mathematical modelling of such systems. In addition, the DF domain is presented in a way that is consistent with the representation of signals and components in the CT and DT domain.

There are several execution models for the DF domain (e.g. concurrent processes, compilation of dataflow graphs, tagged token model) [13]. The most common is to implement dataflow processes as concurrent processes with static scheduling and implement the firing rules as a sequence of "read", "execute" and "write" phases [12,13,15,20,32]. However, these execution models do not match with the signals and components representation of dataflow, as they all use buffers or queues representing channels, i.e. the channel contents, while signals represent instantaneous changes, i.e. channel updates. We will present a new execution model for dataflow, following from representing dataflow models as DF signals and components.

In order to deal with signals as functions of time, our approach uses *higher order functions* to express transformations of signal functions. Hence, we choose for a functional programming language (Haskell) to simulate a mixed-signal system.

3.1 Continuous Time

Consider the following system consisting of a sine source, a time delay and a scope:

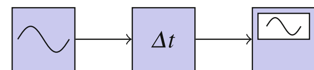

Fig. 1 CT delayed sine wave block diagram

In the CT domain the physical environment of the system or the analogue hardware is represented. In this domain, time is represented by the real numbers, and a signal is represented by a *function* over *all* time. Thus, if f is a signal, then $f(t)$ is the value of that signal at time t. This leads to the following type definitions:

$$Time = \mathbb{R} \tag{1}$$

$$Sig_{CT} = Time \rightarrow \mathbb{R} \tag{2}$$

$$Component_{CT} = Sig_{CT} \rightarrow Sig_{CT} \tag{3}$$

where $A \rightarrow B$ denotes the class of all functions from A to B. Note $Component_{CT}$ is a "higher order type", i.e. it has a function (of type Sig_{CT}) as argument and delivers another function (also of type Sig_{CT}) as result, thus expressing that a component transforms signals.

A component can have multiple inputs and outputs. Multiple inputs and outputs are denoted as tuples (nested ordered pairs). The type of a component with n input signals and m output signals follows as:

$$Component_{CT} = Sig_{CT}^{n} \rightarrow Sig_{CT}^{m} \tag{4}$$

The first component in Fig. 1 is a sine *source* and as such does not really "transform" a signal. That is, it transforms a vacuous input:

$$source\ () = t \mapsto a \cdot sin\ (\omega t) \tag{5}$$

where a is the amplitude, ω the frequency and t is time. The notation $t \mapsto ..t..$ denotes the function which maps t to a (time-dependent) value.

The next component in Fig. 1 is a time delay. The delay can have any value and can even be variable. Existing modelling tools have problems with a time delay because the time of CT signals is discretised for simulation. Mathematically, however, a time delay component is simply defined as (with $\delta = \Delta t$):

$$delay_\delta\ (f) = t \mapsto f\ (t - \delta) \tag{6}$$

where f is a signal, i.e. a function of time, and *delay* adjusts the time of f with δ. Thus, to find the function value on time t *after* a 0.1 time delay, one has to know the function value at time $t - 0.1$. As the input signal of *delay* is a function of time, the time is delayed before f is applied to it. Thus we can locally control or change the time reference of a signal. Note that $delay_\delta$ indeed is of type $Component_{CT}$.

The final component in Fig. 1 is a scope *sink*. The scope plots the signal, hence, as a transformation it may be rather meaningless. That is, the input signal is transformed to a vacuous output, with a plot of the signal as a side-effect:

$$sink\ (f) = t \mapsto () \tag{7}$$

Now suppose a 4Hz sine with amplitude 3 and a 0.1 time delay. The input signal of the sink, that is plotted, is then:

$$delay_{0.1}\ (source\ ()) = delay_{0.1}\ (t \mapsto 3\ \sin (2\pi 4t)) = t \mapsto 3\ \sin (2\pi 4\ (t - 0.1)) \tag{8}$$

The time delay is accurately included in the final function, independent from the time used for simulation. In Simulink and other existing tools it is exactly at this point that inaccuracies are introduced.

3.2 Discrete Time

We extend the system with an analogue-to-digital converter (ADC) and a signal bias in the DT domain:

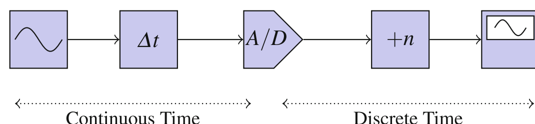

Fig. 2 Mixed CT/DT system block diagram

In the DT domain the digital hardware [such as a finite impulse response (FIR) filter] of a system is represented, in which signals constitute the value of the signal at discrete moments in time. When an ADC is used to sample a CT signal, these values are also called samples.

This leads to the following type definitions:

$$Sig_{DT} = \mathbb{R} \tag{9}$$

$$Component_{DT} = Sig_{DT}^{n} \rightarrow Sig_{DT}^{m} \tag{10}$$

It is important to note that from the perspective of a component a signal now is a *single* value (numerical; here we assume $\mathbb{R}$), i.e. it is a value at a certain time t as in the CT domain but we have no control over t. Note also that the type of components (intentionally) has the same structure as in the CT domain.

A component produces output values dependent on the current input sample and possibly on previous inputs. In order to express the influence of the history of the processing, a component has an internal state which keeps track of the relevant history. Looking at a component as a signal transforming (mathematical) *function* this means that the state has to be modelled as an additional argument to (and result of) that function. However, it is possible to hide the state, here only explained briefly, by directly feeding back the output state leaving only the input signal to be applied to the function. We will discuss this in more detail in Sect. 6.2. Hence, a component can still be seen as a signal transforming function.

Returning to Fig. 2, the ADC component transforms the signal f into a discretised signal with time interval d. This is achieved by flooring the time of the CT input to the latest sample time:

$$adc_d\ (f) = t \mapsto f\ (\lfloor t/d \rfloor \cdot d) \tag{11}$$

Applying the resulting function to a (local) time t gives the latest value that was sampled before t. The output of the adc (which is still a CT function) is then a piecewise horizontal-function. This is implemented efficiently by re-using the results from the latest sample in between sample times.

The next component in Fig. 2 *is* a DT component and adds a constant n to a signal x (as in a bias, or a level shifter):

$$add_n\ (x) = n + x \tag{12}$$

Note that x is a numerical value, in contrast to f (from e.g. the adc_d or $delay_\delta$ specification in the CT domain) which is a function of time.

For a mixed domain model, components from both the CT and the DT domain have to be connected. These components use different types of signals. The output of the ADC gives the latest sample for time t and is a function of time. The add_n component must therefore be changed so it accepts functions of time as input, in order to connect them:

$$\widehat{add_n}\ (f) = t \mapsto n + f\ (t) \tag{13}$$

The notation $\widehat{\ }$ is called "lifting" from a function on values to a function on functions. Lifting changes or embeds the DT component add_n to a CT component $\widehat{add_n}$ in order to combine the component with the CT adc component.

Now suppose a delay of 0.3 time units, a sample period of 0.3 time units and an addition of 1. Then the result of the signal flow diagram (used for plotting) in Fig. 2 is:

$$\begin{aligned} &\widehat{add_1}\ (adc_{0.3}\ (delay_{0.2}\ (source\ ()))) \\ &\quad = t \mapsto 1 + sin\left(2\pi \lfloor (t - 0.2)\,/0.3 \rfloor \cdot 0.3\right) \end{aligned} \tag{14}$$

Note that the time delay and sample time are accurately included in the final function. Thus, the final function combines CT components and DT components in a single expression.

3.3 Dataflow

In the DF domain the software of the system is represented. The DF domain provides a model for stream processing with explicit communication.

In this section we will represent dataflow models as a DF component with input signals and output signals, i.e. the DF domain is represented in the same formalism as the CT and DT domain. This enables the integration of DF models with CT and DT models. Therefore, the environment, the hardware and the software of a system can be represented and simulated in a single unified model.

3.3.1 Processes and Channels

A dataflow model or dataflow process network is a cluster of several independent *processes* that perform computations, and communication between these processes is made explicit via *channels*. A channel is an unbounded first-in first-out (FIFO) *token* container, where tokens are atomic data elements. Processes consume and produce tokens by reading from and writing to channels. The amount of tokens consumed and produced, the rates, can be variable.

3.3.2 Components and Signals

Above, in the CT and DT domain, a component transforms signals, and signals connect components. When applying that approach to the DF domain, a *component* thus corresponds to a node in a dataflow graph, and a *signal* is the data that a component sends (and which consists of a sequence of tokens). This data is then received by another component which stores the tokens in its internal state. As soon as it has enough tokens it will execute, immediately followed by sending the produced tokens.

The above leads to the following type definitions, in which $Token$ is an abstract type to be defined for each application separately (the notation $[Token]$ denotes a list of $Token$s):

$$Sig_{DF} = [Token] \tag{15}$$

$$Component_{DF} = Sig_{DF}^{n} \rightarrow Sig_{DF}^{m} \tag{16}$$

Here too, the structure of a component is the same as before.

Note that the list of tokens does not represent *all* tokens in a channel, but only the currently communicated tokens, similar to the current value in the DT domain. Therefore, the definition is different from the standard implementation of dataflow in which channels store current and previous (unconsumed) tokens and connect processes, while we use signals for connections (current tokens) and store (input) tokens in a component *with* the process. It might seem to the reader that it is easier to represent processes as components and channels as signals. However, this representation does not match well with the semantics of components and signals in the CT and DT domains. The reason is that signals do not have state (memory and state is represented using feedback as will be explained in Sect. 4), but channels are token containers and as such *do* have state. Using signals with state is not a satisfactory option, because as explained in Sect. 3.2 using a mathematical function requires state to be an explicit input and output, i.e. reading a token from a channel involves returning the token and the new state of the channel. Thus, a component reading from channels must also return the new state of all these channels as output. Furthermore, the process writing to this channel needs this updated state of the channel in order to output a once again updated channel state including produced tokens. Clearly, this representation of signals is more complex than just connecting two components.

A better representation of the DF domain is to include the input channel(s) as state of the component. Signals then represent updates, in the form of tokens, to the channels (as channels are unbounded, new tokens can always be added to the channel). Components also implement firing rules, which are directly verified against the number of tokens available in the input channels. This matches well as firing conditions only change with channel updates. Furthermore, the component contains the current production and consumption rates and execution phase as state, needed for determining the firing rules.

Note that according to this definition of DF components, a component transforms a signal each time it receives an update, also in case it has not collected enough tokens to fire (or in case a process has an execution time that has not yet elapsed). To model that, it is possible that a component sends an empty signal containing zero tokens, i.e. a component, applied to an input signal, that does not fire results in an empty output signal. Although an empty signal does not indicate a change to the input channels, it does indicate an execution step so that components update the progressed time represented as execution steps. The other way around, i.e. a signal contains more tokens than a component needs in order to fire, is modelled by allowing a component to execute more than once (if possible), as standard in dataflow models, and to combine the produced tokens (in order) in a single result.

3.3.3 Definitions

Similar to the CT and DT domain, the user specifies the functionality of the component and the connections between them. UNITI takes care of managing channel contents, firing rules and execution.

Consider a DF component that consumes three tokens with a fixed rate, where the tokens are numbers, and calculates their average. The functionality of the process is denoted as:

$$mean_3 \ ([x, y, z]) = [(x + y + z)/3] \tag{17}$$

where the input signal of $mean_3$ is a list of three tokens $[x, y, z]$, and the output signal is a list containing the averaged result as a single token. Clearly, this process can only execute when there are three values available and produces one value. It is thus a process with a token rate of 3 for the input and 1 for the output. Assume an execution time of the process of 1 execution step. Finally, a DF model typically has initial tokens in the channels. Suppose, initially there are two tokens (2 and 4) in the input channel.

The *average* component as a whole, i.e. its functionality together with its initial state, is now formulated as:

$$\begin{aligned} average &= \boxdot\, mean_3 \ \Uparrow \ S \\ &\text{where} \ \ S = ((3, 1, 1), [\,], [2, 4]) \end{aligned} \tag{18}$$

Herein, S is the initial state with the token rates and execution time as the first elements of the tuple, the currently processed tokens as the second element of the tuple (i.e. those in execution), and the input channel contents as the third element of the tuple.

The $\boxdot$ operator and the $\Uparrow$ operator are implemented by the UNITI framework. The above is all that a designer needs to define when using the DF domain. The $\boxdot$ operator is required to add the management of channel contents and firing rules to $mean_3$, so as to embed the functionality defined in $mean_3$ in a DF component *average*. The $\Uparrow$ operator is required to provide the initial state of the component. These operators are discussed in more detail in the next section.

3.3.4 Definitions Provided by UNITI

The complete structure of a DF component is illustrated in Fig. 3. A general component definition is then as follows:

$$comp = \boxdot\, P \ \Uparrow \ S \tag{19}$$

The functionality of a dataflow process is denoted by P. The operator $\boxdot$ extends the functionality of P with firing rules and execution. The $\boxdot$ operator takes care that when the component is applied to a signal i, it will:

- add the tokens from i to its internal state,
- then apply the function P as many times as possible (possibly zero times), each time removing input tokens from the state,
- and finally packing the results in an output signal o.

When a process fires, tokens are consumed from the input channels. However, the output is typically not produced instantaneously, i.e. the execution time of a process is also taken into account.

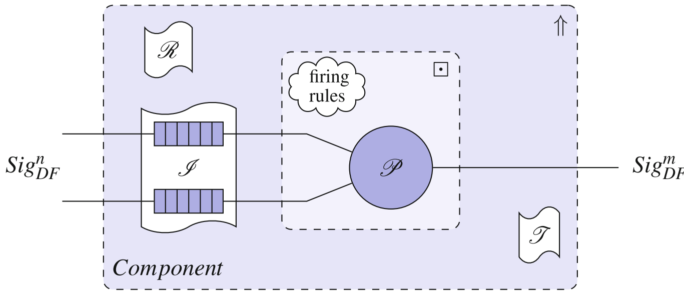

Fig. 3 DF component structure

The functionality of a DF process P of type $\mathscr{P}$ is a function from inputs to outputs:

$$\mathscr{P} = \mathscr{I}^n \rightarrow \mathscr{O}^m \tag{20}$$

Note that the input and output of P are of the same type as the inputs and outputs of the DF component:

$$Component_{DF} = Sig^n_{DF} \rightarrow Sig^m_{DF} \tag{21}$$

However, for the component the signals are channel *updates*, while for the process P they are the r_i input tokens and r_o output tokens of the process.

The (unary) $\boxdot$ operator converts $\mathscr{P}$ to a function on input signals and state. So the type of $\boxdot$ is:

$$\boxdot : \mathscr{P} \rightarrow \left(\mathscr{S} \times Sig^n_{DF} \rightarrow Sig^m_{DF} \times \mathscr{S}\right) \tag{22}$$

Herein, the resulting function has a current state (S of type $\mathscr{S}$) and input channel updates (i of type Sig^n_{DF}) as inputs, and returns the output channel updates (o) and the updated state (S') as outputs.

After applying the above function to an initial state and an input i, it returns output o and the next state S'. The function is then already applied to next state S' using so-called *partial application* while the next input will follow later. The result is then an output o and a new function to use for the next input i. By repeating this each time, the state is not visible from the outside but hidden in the next function to use. This is performed by the (binary and infix) $\Uparrow$ operator:

$$\Uparrow : \left(\mathscr{S} \times Sig^n_{DF} \rightarrow Sig^m_{DF} \times \mathscr{S}\right) \times \mathscr{S} \rightarrow Component_{DF} \tag{23}$$

Thus, the $\Uparrow$ operator applied to the result from the $\boxdot$ operator applied to P, and the initial state S result in a DF component $Component_{DF}$.

Implementation details can be found in [21]. In our implementation the firing rules, i.e. the token rates required for execution, of the dataflow process are generalised to support the different dataflow classes from single-rate dataflow (SRDF) to dynamic dataflow (DDF). The remainder of the implementation is firing rule agnostic.

3.4 Representation in Haskell

The mathematical formulations of the CT, DT and DF domain can be straightforwardly represented in a functional language. In particular the support for higher-order functions to express signal transformations, and partial application (a function is already applied to part of the arguments, while the rest follows later), that functional languages offer, are essential. Furthermore, side-effects are not allowed in any of the domains, so we choose for the pure (side-effect-free) functional language Haskell. Haskell also provides a *type class* feature to conveniently overload algebraic and composition operators, so that the same operators can be used in all domains. That means that the type of the signal determines the specific operator implementation that is used.

We will first present the representation of the *delay* component as presented for the CT domain, and the *adc* and *add* components as presented for the DT domain. The *delay*, *adc* and *add* components are straightforward reformulations of Eqs. (6), (11) and (12) as can be readily checked (`\t ->` corresponds to $t \mapsto$ and the standard function `floor` returns the greatest integer not greater than x):

```
delay delta f  =  \t -> f (t-delta)
adc   d     f  =  \t -> f (floor (t/d) * d)
add   n     x  =  n + x
```

Note that these functions have two arguments, one of which was given in the form of a subscript in the mathematical formulation. In the Haskell formulation the first is given when used as a component, while the second (`f` and `x`) represent the input signals which follow when a component is composed with another component. Note also that a space in Haskell represents application of a function to an argument and has the highest precedence, i.e. $f(1)$ is denoted as f 1.

For implementing the DF domain, the `df` operator is the Haskell formulation of the $\boxdot$ operator and `^^^` is the formulation of the $\Uparrow$ operator. The DF component *average* [Eq. (18)] is then implemented as:

```
average = df mean_3 ^^^ ((3,1,1),[],[2,4])
```

4 Unified Composition

The standard mathematical interpretation of a signal flow diagram is that the arrows express *signals* and the components denote *signal transformations*. Furthermore, the diagram as a whole then is a *composition* of these transformations.

For each of the domains we have seen different kinds of signals:

- Signals in the CT domain are functions of time, i.e. they represent the value of the signal over all time.
- Signals in the DT domain are values at discrete moments, also called samples.
- Signals in the DF domain are lists of tokens.

An important observation is that processes of a dataflow model also transform data. Therefore, components in the DF domain can also be represented as *signal*

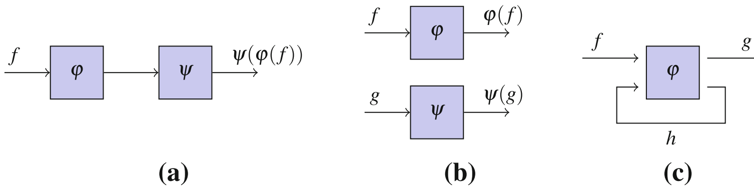

Fig. 4 Composition operations. (**a**) Sequential, (**b**) parallel, (**c**) feedback

transformations. The perspective on the data, the representation of the signal, that is transformed in each domain is fairly different, as we have seen. Yet, in *all* domains components are signal transformations. This is intentional, so we can provide generic rules for composition. The generic component structure is defined as:

$$Component = Sig^n \rightarrow Sig^m \tag{24}$$

Note that Sig^n can be any combination of signals from the various domains and is implemented as a nested tuple, e.g. $(Sig_{CT}, (Sig_{DT}, Sig_{DF}))$.

Next, we will define operators for sequential, parallel and feedback composition. With these composition operators we can define arbitrary signal flow diagrams [27].

4.1 Sequential

Sequential composition (illustrated in Fig. 4a) is defined as:

$$\varphi \triangleright \psi = f \mapsto \psi\ (\varphi\ (f)) \tag{25}$$

where φ and ψ are transformations, i.e. components, and $\triangleright$ is the operation to compose transformations sequentially. That is, $\varphi \triangleright \psi$ is the transformation that takes a signal function f as an argument and determines its result by first applying φ to f and then ψ to the resulting signal function. Thus, $\triangleright$ returns a new component with the input signal f of φ and the output signal of ψ.

As an example, consider a sine source that is accelerating into the direction of an (stationary) observer which causes a change in the observed frequency; the Doppler effect. The resulting signal increases in frequency with time, which is modelled with time scaling in the CT domain as:

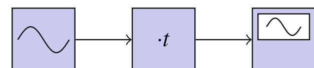

Fig. 5 Accelerating source block diagram

and denoted as:

$$source \triangleright tscale_a \triangleright sink \tag{26}$$

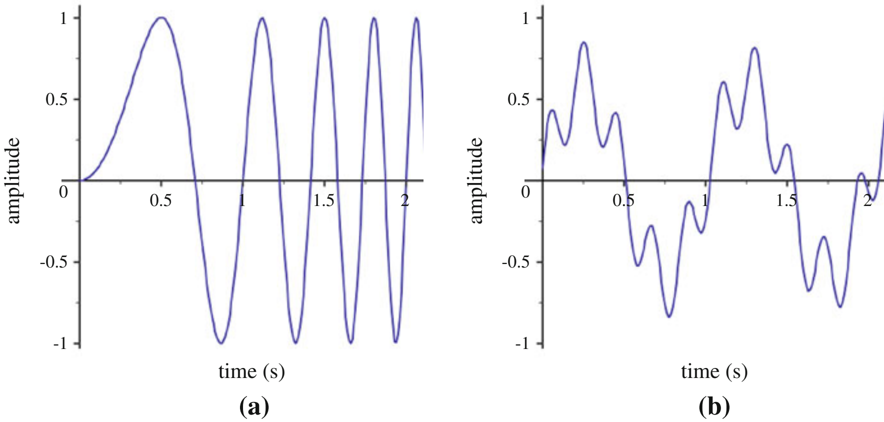

Fig. 6 Simulation results of the sequential and parallel composition examples. (**a**) Chirp signal waveform, (**b**) two added sources waveform

The $source$ and $sink$ were defined in Sect. 3 and are repeated here for clarity. The definition for $tscale_a$ is:

$$source\ () = t \mapsto sin\ (t) \tag{27}$$

$$tscale_a\ (f) = t \mapsto f\ ((a \cdot t) \cdot t) \tag{28}$$

$$sink\ (f) = t \mapsto () \tag{29}$$

As before, these definitions can be straightforwardly represented in Haskell, and evaluated for simulation. The simulation result is shown in Fig. 6a and as expected shows a frequency that increases over time.

It is sometimes useful to connect a single output of a component to multiple inputs of other components, thereby duplicating the signal. Since the number of inputs to connect to is known from the context of the sequential operator, we can define a generic definition:

$$\varphi \rhd^* \psi = f \mapsto \psi\ (g, g, \ldots), \quad \text{where} \quad g = \varphi(f) \tag{30}$$

Herein, φ is applied to the input signal f, and the resulting output signal g is used for as many input signals of ψ as needed.

4.2 Parallel

Likewise, parallel composition (Fig. 4b) is defined as:

$$\varphi \parallel \psi = (f, g) \mapsto (\varphi\ (f)\,, \psi\ (g)) \tag{31}$$

i.e. multiple inputs are represented as tuples and the composition connects φ to the first and ψ to the second. Note that for CT components each input and output in a (parallel) composition is an independent function of time, and can thus have a different time reference (also see Sect. 5). Only when two signals are combined, for example with an addition, are the time references of the input signals (locally) made the same.

As an example, consider the following system:

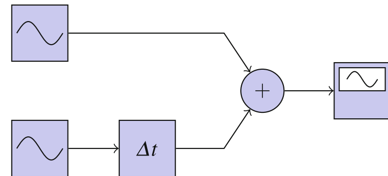

Fig. 7 Simple beamformer block diagram

The system consists of two sources with different delays which are added, corresponding to a simple beamformer (see Sect. 8). This system is denoted as:

$$(source_1 \parallel (source_2 \triangleright delay_\delta)) \triangleright (+) \triangleright sink \tag{32}$$

The simulation results are shown in Fig. 6b.

An alternative definition is:

$$system = (\varphi_1 \parallel \varphi_2) \triangleright (+) \triangleright sink \tag{33}$$
$$\text{where} \tag{34}$$
$$\varphi_1 = source_1 \tag{35}$$
$$\varphi_2 = source_2 \triangleright delay_\delta \tag{36}$$

By using a `where` clause, we introduced hierarchy, i.e. in the definition of *system* we define two blocks to be in parallel (φ_1 and φ_2) and in the where-clause we define what these blocks are. So structural hierarchy is easily achieved by naming subsystems. This is possible because a composition of components is itself a component.

It is sometimes useful to connect a duplicate of a single component to each of the input signals in parallel. Since the number of inputs known from the context of the parallel operator, we can define a generic definition:

$$\parallel^* \varphi \;=\; \varphi \parallel \varphi \parallel \ldots \tag{37}$$

Herein, $\parallel^*$ creates as many duplicates of of φ (in parallel) as needed.

4.3 Feedback

In many systems there is a feedback loop in the system. The component φ in Fig. 4c shows two inputs f and h and two outputs g and h. The signal h forms the feedback loop, i.e. the second output of φ is also used as input. From the outside, the resulting

component only has an input signal f and an output signal g. However, g is determined by applying φ to f and h, where h is also determined by applying φ on f and h resulting in a recursive dependence. Thus, at some point φ must be able to determine h at the output using only f. Feedback composition (Fig. 4c) is then defined as:

$$\circlearrowright \varphi = f \mapsto g, \quad \text{where} \quad (g, h) = \varphi\ (f, h) \tag{38}$$

Herein $\circlearrowright$ connects the second output of φ to its second input, thereby returning a component with input f and output g.

As an example we take an RC low-pass filter:

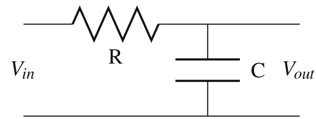

Fig. 8 RC low-pass filter

Applying Kirchhoff's current law we get:

$$\frac{V_{in}\ (t) - V_{out}\ (t)}{R} = C \frac{\mathrm{d}V_{out}\ (t)}{\mathrm{d}t} \tag{39}$$

which we can rewrite to:

$$V_{out}\ (t) = \frac{1}{RC} \int_{-\infty}^{t} (V_{in}\ (t) - V_{out}\ (t))\ \mathrm{d}t \tag{40}$$

and which corresponds to the following block diagram:

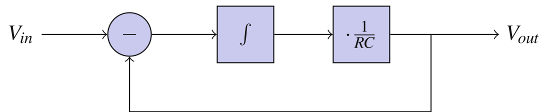

Fig. 9 RC filter block diagram

This block diagram corresponds to a component with V_{in} as input signal and V_{out} as output signal. Furthermore, it has three sub-components ($-$, $\int$ and $\cdot\frac{1}{RC}$) which are sequentially connected and of which the result is fed back to the input. Hence, it is defined in UniTi as (with id the identity function):

$$filter_{RC} = \circlearrowright \left((-) \triangleright \int \triangleright \left(\cdot \frac{1}{RC} \right) \triangleright^{*} (id \parallel id) \right) \tag{41}$$

4.4 Integration

So far, we have discussed how to define components in the various domains and how to compose components within each domain. Now we will discuss how to compose mixed CT, DT and DF components for a multi-domain simulation. This is achieved by embedding a DF component in a DT component and a DT component in a CT component such that for simulation purposes the CT domain is the unifying domain.

4.4.1 DT ⇒ CT

To embed a DT component into a CT component it must accept a function of time instead of a single value (see Sect. 3), i.e. we have to "lift" the DT component to a CT component. In Sect. 3.2 we introduced the notation $\widehat{add_n}$ for the "lifted" version of add_n. Here we will generalise that notation into a *lifting operator* $\widehat{\ }$.

For unary functions (say g) the operator $\widehat{\ }$ is defined as follows:

$$\widehat{g}\ (f) = t \mapsto g\ (f\ (t)) \tag{42}$$

whereas for binary operations (say h) it is defined as follows:

$$\widehat{h}\ (f, g) = t \mapsto h\ (f\ (t)\ , g\ (t)) \tag{43}$$

Clearly, this can be generalised immediately for n-ary functions. So, when a designer has a DT component C in his design, he can simply replace it by $\widehat{C}$ to get a CT component. This lifting is automatically performed by the sequential composition operator if the component on one side is in the CT domain and on the other side in the DT domain (see [21] for details).

As an example, let us revisit Fig. 2 which is defined as (with a delay of 0.15 time units, a sample period of 0.08 time units and an addition with 0.25):

$$source \triangleright delay_{0.15} \triangleright adc_{0.08} \triangleright add_{0.25} \triangleright sink \tag{44}$$

The simulation results are shown in Fig. 10a, showing a delayed, sampled and biased sine wave as corresponding to the definition of the mixed-domain model.

4.4.2 DF ⇒ DT

DF signals are a list of tokens, while DT signals are values. Thus, to embed a DF component in a DT component, it must accept single values (samples) instead of a list of tokens. This presents a problem, because in DF models data is abstracted away into tokens, i.e. tokens are arbitrary data, while samples are values. We could also abstract from data in the CT and DT domains; CT signals would then be functions of time to tokens, but these have no sensible physical representation. So instead we limit embedded DF models to values as tokens at the boundaries. This is achieved by writing the value from the DT domain as a token into the input channel of the DF component at the sample time. So a value from a DT signal is converted to a DF signal

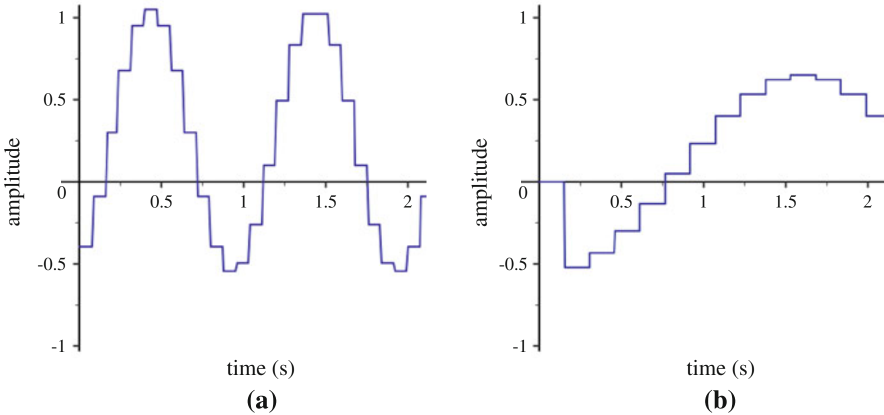

Fig. 10 Simulated results of a mixed CT/DT and a mixed CT/DT/DF domain model. (**a**) Delayed, sampled and biased sine wave, (**b**) averaged sine wave over three samples

consisting of a singleton list with that value as token, i.e. from the perspective of the dataflow model, the DT domain produces single tokens at a fixed rate. Vice-versa, a single token output DF signal is converted to a DT value (possibly with a delayed sample time because of the execution time). Note that practically this is not a very serious limitation as we can easily include an additional dataflow process that converts the single input tokens from the source into more complex tokens, or a dataflow process that convert more complex tokens into single tokens for a sink. Note also that dataflow models typically represent signal processing applications where the sources and sinks already correspond to ADC- or digital-to-analogue converter (DAC)-like components.

4.5 Unified Model

Now components from all domains can be composed (by taking the DT domain as an intermediate step in case of a DF component). An example of a mixed domain system then follows as:

$$source \triangleright delay_{0.2} \triangleright adc_{0.05} \triangleright add_{0.15} \triangleright average \triangleright dac \triangleright sink \tag{45}$$

where $source$, $delay_{0.2}$, $adc_{0.05}$, dac and $sink$ are CT components, $add_{0.15}$ is a DT component, $average$ is a DF component, and their composition $system$ is a CT component. The simulation results are shown in Fig. 10b. The $average$ component averages three samples, as defined in Sect. 3.3, and therefore only outputs a token every three input tokens or samples from the DT domain. In between no output is provided. The dac components holds the latest output (sample), and the plot function of the $sink$ connects those point with lines, as shown.

5 Unified Time

Of major importance is the interaction of time in multi-domain models; components need a uniform view of time to be able to correctly interact. Indeed, how to integrate time is an acknowledged problem [10,11,33]. Yet, this integration does *not* require a global notion of time. UniTi provides a novel unified perspective on time for multi-domain models, in which different notions of time are differentiated. Moreover, time is local, allowing exact time transformations, integration of time over domains, and local solvers which use different precision and complexity.

5.1 Notions of Time

In the multi-domain models of Sect. 4 we identify different notions of (modelled) time:

- the instants when the designer wants to know the behaviour of the system, the *simulation time*,
- the instants when continuous information from the environment is sampled by, say, an ADC, the *sample time*,
- the time steps that are necessary to numerically approximate functions (e.g. an integral), the *approximation time*,
- the time that has elapsed during processing, the *execution time*,
- the time locally, possibly transformed by e.g. a time delay, the *local time*.

The last notion of time is necessary to represent relativity: different distances from a source lead to different local time references relative to the source. From the perspectives of components at different distances, the source is at a different time, yet the source is defined for a single time reference. Therefore, each component at a certain distance must have its own local time reference to the source, i.e. time is a local property and time is relative. This occurs for example for a front-end with multiple signal paths, which might have slightly different path lengths, thereby modelling non-ideal common mode noise rejection.

5.2 Time Transformations

Existing tools perform a simulation by extracting a set of equations from the model. Then the set of equations is (in the general case) solved numerically, i.e. solvers numerically approximate the differentials. Such solvers operate iteratively with a fixed or variable step size. Each step the equations are evaluated and the results updated according to the algorithm. This iteration step is a time step, i.e. iteration is performed over time. Time is thus a global property of the model as it is applied to the complete equation set, which represents the complete model. Furthermore, the same solver is used for all components in the model for the same reason.

A time transformation such as a time delay or time scaling receives input values at the current time step, but must output values from another time step. Therefore, such a time transformation component buffers time-value pairs. Note that this also

restricts time transformations to only depend an earlier time. Furthermore, as the time transformation can be variable and in general unpredictable, it can not be guaranteed that the exact time-value pair that is needed is buffered. As such the output value is interpolated from known time-value pairs. This interpolation introduces inaccuracies in the resulting value. Of course, these inaccuracies can be reduced by reducing the step size, however this makes the simulation less efficient.

In UNITI the CT signals are *functions of time* and components such as the time delay presented in Sect. 3.1 are signal transformations. Therefore, the time delay is accurately included in the final function, independent from the time used for simulation. In other words, the equation to use for simulation is *build* by the composition operators.

Multi-rate systems contain DT domain samples that are generated by ADCs operating at different rates. Such systems are typically problematic for simulation because the data must be aligned with a global clock tick. Thus if we consider the samples of the ADC over time as a list, then in multi-rate systems the positions in the lists correspond with different times, which are difficult to merge. As we have separated these notions of time, this is not a problem in UNITI.

Consider, for example, a multi-rate mixed-signal model with an ADC with rate 0.3 and an ADC with rate 0.35. This system is defined as follows:

$$(ramp_{0.2} \triangleright adc_{0.3} \parallel ramp_{0.3} \triangleright adc_{0.35}) \triangleright (+) \triangleright sink \tag{46}$$

$$ramp_r \; () = t \mapsto r \cdot t \tag{47}$$

A simulation at time 2, means that the ADCs should output the latest samples, which are the samples from time 1.8 and 1.75 respectively. Thus at the ADC we floor the simulation time to the last sample time and use that to evaluate its CT input signal.

5.3 Time Integration

The different domains in particular have a different view of time. Integration of the domains is therefore all about integrating the meaning of time in each domain. We have already shown how DF components are embedded in DT components and DT components in CT components for multi-domain simulations. Here we will discuss the consequences for time in each domain in more detail.

5.3.1 CT

The signal representations for each domain are a conscious choice. For a CT component, the input signal represents a function over all time. From the perspective of a CT component the input signal as a whole over all time is transformed, there is no discretisation of time needed for implementation purposes. Therefore, its time reference can be changed, the signal can be delayed or time can be scaled (e.g. speed-up).

5.3.2 DT

Yet, from the perspective of the DT component, the input value x is just a value at a discrete moment in time. This representation is deliberate as a DT component should not have control over time, as the digital hardware it represents does not either; it depends on the current sample and possibly on previous samples and nothing in between; features such as a combinatorial delay is a property of the component, not of the component functionality. From the perspective of the DT component, the input signal is just a value, but a value that changes over time. The DT component can not and should not be able to influence the time at which a value changes. A common option is to have a DT component operate on a list of values over all time, but this is not desirable because it would give the DT component access to all future and past values. Only having access to the current value forces a DT component to make its state explicit, i.e. state is an explicit input and output of the component. As such there is only a notion of the next value. Thus, the state and output of an operation change with a new input value, irrelevant at which time this is. From the perspective of the component, time is abstracted away to discrete instants.

As a result, there is no time in the DT component definitions. When such a DT component is lifted to a CT component, it still has no access to time t as the lifting is performed on the original definition. In other words, the lifted version performs the DT functionality on the value of the CT input signal at some time t which corresponds to the sample time for that component. DT signal transformations are not applied for all time, but only at the sample times. The boundary between the CT and DT domain is the ADC, which samples the CT domain to provide that value to the DT domain. Thus from a CT perspective, the ADC floors the time to the latest sample time and holds that value until the next sample time. So the composition of a CT component with a DT component is a CT component, but one that only evaluates and returns the result at the latest sample time for any time in between.

For a simulation, we are interested in the behaviour of the system, i.e. the results of the model over time. Therefore, simulation is a CT process, i.e. we evaluate the model over time. As such, it makes sense to embed a DT model in a CT model for simulation. This is still efficient, because time is floored to the latest sample time (as with the ADC) and the sample value is re-used for every evaluation using state.

5.3.3 DF

The DF domain is untimed and only models the ordering of tokens. In the DT domain samples are linked to a sample time. So from the perspective of time, which is of primary importance in a simulation model, a DT model contains more information than a DF model and it makes sense to embed a DF model in a DT model. Because the DT values are linked to a sample time, the DF process now is extended with time. The production time of a token from a source process is the sample time of its DT value. After the execution time of the processes in the DF part of the model, the production time of a token from a sink process is the sample time of that (token) value in the DT. Therefore, execution time of a DF process also has meaning; the sample times in the DT domain are delayed by the execution time.

5.4 Local Solvers

For existing tools, the inputs and outputs of the model components are values at a certain global time step. This means that the time step determined by the solver for numerical approximation of a differential equation is applied to all equations. The global time step causes the whole system to be evaluated, while it is very well possible that most of the system does not need to be evaluated at this fine-granularity, thereby reducing efficiency; for example, the DT domain most likely has a sample period much larger than the approximation step of e.g. an integral in the CT domain, which often has to be very small for sufficient accuracy.

In UniTi time is kept local, i.e. every continuous component may have its own discretisation of time in time steps. Thus, components may be numerically calculated at a fine time scale without causing inefficiencies in those parts of the system which do not need such a fine time scale.

Such time steps are typically needed to approximate operations that change *over* time, such as integration. We can solve the integral (or a differential) symbolically or numerically. A problem with symbolic integration is that for many functions an analytical solution does not exist. Thus, simulation tools solve the general case with numerical integration. Haskell has good possibilities to define analytical solutions to integral definitions whenever possible, but we will use numerical integration for generality, as in the standard approach in simulation tools.

The component for integration is defined as:

$$\int (f) = t \mapsto \int_{t_0}^{t} f(t)\ dt \tag{48}$$

Numerical integration relies on a recurrence relation to approximate the integral. A simple numerical integration method is the Euler method:[1]

$$y_t = y_{t_0} + \sum_{0}^{n-1} h \cdot f(t_n), \quad \text{where} \;\; n = (t - t_0)/h, \;\; t_{i+1} = t_i + h \tag{49}$$

Here, h is the approximation time step that is used locally. For this definition, t determines the number of steps n to compute from time t_0 using time step h. Furthermore, t_n is the time for step n at which to evaluate input signal f. The Euler method can also be represented with a feedback composition operator (↻):

$$\int = (\cdot h) \triangleright \circlearrowright \big((+) \triangleright^{*} (id \parallel delay_h)\big) \tag{50}$$

[1] Of course we can also use more sophisticated numerical algorithms such as Runge–Kutta, which determines the time step based on the tolerance in accuracy of the result.

where id is the identity function and $\triangleright^*$ duplicates the input signal, which is delayed. The delayed signal is fed back to one of the inputs of the addition, i.e. one of the arguments is a delayed version of itself, which recurses back until the initial value.

Note that for each use of the integral component a different approximation step can be chosen by the designer. The accuracy of the approximation depends on the correspondence between the dynamics of the signal and the step size. As this can differ at different places in the system, it is very useful to be able to define the time resolution per integral (or differential). However, to the best of our knowledge, all simulation tools use a single implicit time step to update the whole system, therefore potentially unnecessarily calculating simulation results for much of the system. We conjecture that current tools use a global simulation time step, because it is difficult to determine the different time steps at each place in the system. However, with our approach, by locally applying the time steps, the time used for evaluation is only propagated back to the input signal, i.e. only the input signal and the components that determine that input signal are evaluated using the local time step. So, signals at the input are evaluated each approximation time step, but how often the result at the output and the following blocks is evaluated is not influenced.

In these implementations, the integral is recalculated from time t_0 to t each simulation step. A more efficient implementation that re-uses previously calculated values needs state. State is discussed in Sect. 6.2.

6 Simulation

Simulation of a model consists of evaluating the model and visualising results. A major difference between our approach and other mixed-signal modelling tools (see Sect. 2) is the way the model is simulated. Our approach is based on function composition, while other tools are based on value-passing between components.

6.1 Evaluation

As models are a composition of functions, simulation is simply a matter of evaluating the composed function or model. Since simulation is evaluation over time, the top level component needs to be a CT component. The output of the top component is a function of time to a vacuous result, of which the time is used as a time step for updating the visualisations, i.e. the model is updated until that time. In addition, the top component has a vacuous input.

The visualisation update or simulation time step can be larger than the time step used locally e.g. for sampling or an integral approximation. In that case, the ADC or integral component are evaluated a few times with their local time step until they have reached the simulation time. For example a plot component uses its local time step to evaluate its input signal. The time thus propagates backwards through the components. Only when necessary, for example for numerical approximation, are the input signals evaluated with smaller time steps, and only at the input of the component.

As an example, consider the model from Sect. 4.4.1 repeated here:

$$source \triangleright delay_{0.15} \triangleright adc_{0.08} \triangleright add_{0.25} \triangleright sink \tag{51}$$

Assume a sine source, i.e. $source\ () = t \mapsto \sin(t)$. The input signal of $sink$ which is used for visualisation is then:

$$add_{0.25}\ (adc_{0.08}\ (delay_{0.15}\ (t \mapsto \sin(t)))) \\ = t \mapsto 0.25 + \sin(\lfloor (t - 0.15)/0.08 \rfloor * 0.08) \tag{52}$$

As explained before, the important aspect from this example is that t is unaltered by add, floored by adc and shifted by $delay$ before sin is applied to it.

6.2 Memory and State

The use of memory or state has been mentioned several times. DF components have state for remembering the input channel contents. An example of a DT component with state is a FIR filter. A FIR filter uses a history of recent inputs for calculating the current output. An example of a CT component with memory is an integration.

Essentially, memory or state is not necessary for correct simulation results. Previous inputs (in case of DT) or an approximation over time (in case of CT) can simply be recalculated every time they are needed. However, simulation would quickly become very inefficient because of these redundant calculations. If we restrict t to be totally ordered, we can re-use previously calculated results. So for reasons of efficiency, previously calculated results are remembered.

The difference in performance between a simulation with state and without state is quite substantial, especially for multiple integrations in sequence. For example, simulating the RC-filter system from Sect. 4.3 with 150 simulation time steps, and with about 10 integration steps per simulation time step, takes 2.833 s without state and 0.119 s with state on a 2 GHz Core 2 Duo system (a 24× speed-up). For two integrations in sequence, simulations without state almost become unmanageable taking 1863.201 s (∼30 min) against 0.718 s with state (a 2,600× speedup).

The functionality of a component with state is defined with a function that has an extra input and an extra output for the state. When composing such functions, the state output of a component must be separated from the normal output, because the state must be fed back to the component itself while the other outputs are connected to other components. Clearly, there is a lot of additional effort managing the state that is not related to designing the system. More problematic, however, is that the composed component has a state containing all the state of its sub-components, i.e. state hierarchically moves up all the way to the top level component and for the top level component, the state of all sub-components is combined in a single state. This state has to be packed in and out, at each level, and each time a function with state is used. Therefore, state is globally managed, while it is a local property of a component. What we would like is to provide an initial state to the component and keep it local,

i.e the state should not be visible when composing components. This can be achieved by making use of continuations to hide the state, as discussed next.

6.2.1 Continuations

There are several options for implementing state hiding. We choose to use continuations, because it matches well with our representation for components and composition. A continuation represents a function, or in our case a component, that is to be used for the next input, i.e. it represents a function to continue the computation for the next input. Using continuations, the composition operators can hide all the details for supporting state from the user.

To apply this, a component is a function from an input to an output *and* a new version of itself with updated state, the continuation:

$$Component = Sig^n \rightarrow \left(Sig^m, Component\right) \tag{53}$$

As discussed above, the functionality of a component is defined using a function with an explicit input and output for state:

$$f : \mathscr{S} \times \mathscr{I} \rightarrow \mathscr{O} \times \mathscr{S} \tag{54}$$

$$f\ \ (s, i) = \left(o, s'\right) \tag{55}$$

This function is applied to an initial state s_0 and an input and returns an output and a new state s'. Thus, the new state is already available before the next input. If f is partially applied to s', we get a new function f':

$$(o, s') = f(s_0, i) \tag{56}$$

$$f'(i) = f\left(s', i\right) \tag{57}$$

where f' is the continuation, i.e. the function to use for the next input.

This principle is recursively applied for each new input using the $\Uparrow$ operator, which is defined as:

$$f\ \Uparrow\ s = i \mapsto \left(o,\ f\ \Uparrow\ s'\right) \tag{58}$$

$$\text{where}\ \ (o, s') = f(s, i) \tag{59}$$

When a component is defined the initial state is provided ($f\ \Uparrow\ s_0$), after that each input results in an output o and a continuation ($f\ \Uparrow\ s'$), with s' the next state. At the outside of $f\ \Uparrow\ s_0$ the state is not visible. The $\Uparrow$ combinator was already presented for the DF domain in Sect. 3.3.

Of course, now we have to manage the continuations instead of the state. However, in contrast to the state, the continuations do compose. This is performed automatically by the composition operators. Thus, components with state compose just as

components without state using these operators. All that remains are definitions for the composition operators:

$$\varphi \rhd \psi = f \mapsto \left(h,\ \varphi' \rhd \psi'\right), \tag{60}$$

$$\text{where} \quad \left(g,\ \varphi'\right) = \varphi\ (f)\,,\ \left(h,\ \psi'\right) = \psi\ (g) \tag{61}$$

$$\varphi \parallel \psi = (f, g) \mapsto \left(\left(f', g'\right),\ \varphi' \parallel \psi'\right), \tag{62}$$

$$\text{where} \quad \left(f', \varphi'\right) = \varphi\ (f)\,,\ \left(g', \psi'\right) = \psi\ (g) \tag{63}$$

$$\circlearrowright \varphi = f \mapsto \left(g,\ \circlearrowright \varphi'\right), \tag{64}$$

$$\text{where} \quad \left((g, h)\,, \varphi'\right) = \varphi\ (f, h) \tag{65}$$

Herein, for $\rhd$, component φ is applied to input f, resulting in an output g and a continuation φ'. For ψ, it is applied to g resulting in h and a continuation ψ'. The result is a component with input f resulting in output h and a continuation: the sequential composition of the continuations of the sub-components.

Multiple components with state can be composed using the composition operators. Components without state need to be represented in the same form as Eq. (53) in order to compose with component with state. This is easily achieved using:

$$\uparrow (f) = i \mapsto (f(i),\ \uparrow (f)) \tag{66}$$

where the continuation is just the same function f again.

7 Model Transformations

So far, the presented framework allows for a single model with multiple domains when modelling, designing, and simulating systems. Another identified aspect of model-based design is iterative refinement. Typically this is performed manually by the designer, as current tools lack model transformation support (see Sect. 2). Because of the integrated approach presented with UNITI, we can apply model-based design using transformation steps. These model transformations, guided by the designer, automate the refinement of the model. Further, the transformations preserve correctness of the design and are reversible, thereby enabling design space exploration.

We identify three important steps in the design process: co-design, partitioning, and mapping and code-generation. Although these steps by themselves are not new, matching them with the presented domains as model transformations is. This is not a trivial connection, since, as we found in Sect. 2, what is typically understood as model transformations is only code-generation. Code generation does not allow design space exploration and true iterative refinement as it is a one-way process.

Partitioning in the case of software is also called parallelisation. Because we use the DF domain to model (concurrent) software, dependencies are explicit. Parallelisation is then straightforward, but becomes highly dependent on the way the algorithm is specified and implemented. We will show dataflow directly provides control parallelism and that it is beneficial to represent algorithms as operations on aggregate data for

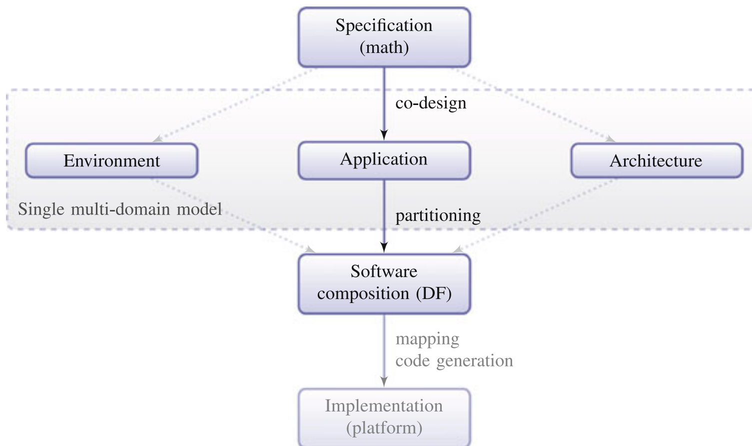

Fig. 11 Design flow for tiled architectures

data parallelism, because we can then define model transformations on the definitions using higher order functions. Being able to apply such model transformations to a single model for multiple domains is a unique advantage of UniTi. For more detail on the kind of transformations that can be performed with such an approach, we refer the reader to [28]. Here we will present only the dot-product as an example.

Next, we will present the design flow and the design steps in more detail.

7.1 Design Flow

The design flow, focusing on software refinement for a multi-core tiled architecture, is illustrated in Fig. 11. The rounded rectangles represent models and the arrows represent transformations. A single multi-domain model includes the environment, the architecture (analogue and digital hardware) and the application (software). The design flow uses a top-down divide-and-conquer approach. The initial (formal) specification of a system is readily implemented and verified in the CT domain. Co-design can be seen as a division *over* the domains, while partitioning can be seen as a division *within* a domain. The mapping and code generation step are beyond the scope of this paper.

7.2 Co-design

During the co-design process, functionality is divided over the different domains. Co-design emphasises that the different perspectives in the domains are part of the system design and need to be included. We distinguish a number of tasks:

Cyber/physical co-design Decide what is needed from the environment for simulation and verification of the system. The environment is modelled in the CT domain.
Analogue/digital co-design Define the architecture and decide what is implemented in analogue hardware (CT domain) and what in digital hardware (DT domain).

Hardware/software co-design Decide what to do in fixed hardware (ASIC, FPGA) and what to do in programmable hardware and software (DF domain), thereby refining the architecture and defining the application.

UniTi supports a number of features to assist the co-design step of the design flow. The basic algebraic mathematical operators such as $+$ and $\cdot$ are overloaded so the same operator can be used for all domains using the *type class* feature of Haskell. That means that the type of the signal determines the specific operator implementation that is used and that the semantics of the operator in each domain are the same.

For example, a mixed CT and DT model is transformed from a CT model by only adding an ADC. A domain independent definition of an addition of 1 (bias) followed by a multiplication with 0:12 (gain) is:

$$(+1) \triangleright (*0.12) \tag{67}$$

where the input signal determines whether functions of time, values, or tokens are added and multiplied. However, without changing the definitions and by only adding an ADC, the addition is in the CT domain while the multiplication is in the DT domain:

$$(+1) \triangleright adc_{0.1} \triangleright (*0.12) \tag{68}$$

Of course, the placement of the ADC and in general the division over the domains is a manual operation by the designer, as the relevant properties for assessing the trade-off, such as cost in terms of money or energy, are not part of the model. That is not to say they could not be; further research into this direction would be interesting.

7.3 Partitioning

For the partitioning step we will focus on partitioning the software, i.e. parallelisation.

We identify two kinds of parallelism: data parallelism and control parallelism. In the first kind of parallelism the data is split. Examples are bit-level and data-level parallelism. In the second kind the control, i.e. the operations on the data, is split. Examples are instruction, task and pipeline parallelism.

7.3.1 Control Parallelism

Control parallelism occurs when some operations are independent or functions are executed in sequence. A section can already continue with the next data, while later sections are still operating in parallel on previous data. To keep execution functionally correct, the sections may not influence each other besides the explicit input and outputs, i.e. the function must be side-effect-free with respect to the calculation.

These restrictions are captured by the DF model. Passing arguments to mathematical functions is similar to communicating values between processes. In the DF domain, data in channels must remain ordered, making sure the operations are performed in sequence. Back-pressure (a process is stalled if the tokens are not consumed from the output buffer fast enough by the next process) ensures automatic synchronisation

in parallel execution. Thus, the computation (functionality) and communication (the inputs and the outputs) are made explicit to fit to the dataflow model and are wrapped in a DF component.

7.3.2 Data Parallelism

Data parallelism occurs when some operation or function has to be executed on the data in aggregate data structures such as lists, arrays or trees. There are at least two elementary forms of such operations, the first applies an operation to each element of an aggregate data structure separately, the second gathers the elements together into a single outcome (as in "map-reduce"). The dot product below explains this in further detail.

7.3.3 Aggregate Operations

In order to recognise and isolate data and control parallel properties of operations in an application, it is beneficial to formulate the application on a level that is as high as possible. That is to say, to specify operations on the aggregate level rather than on the element level.

As an example, consider the standard definition of the dot product of two vectors (such as used for a FIR filter or beamformer):

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{N} a_i \cdot b_i \tag{69}$$

In this definition the operations for addition and multiplication occur on the element level, where the individual elements are indicated by the index i. This formulation strongly suggests a *for*-loop in which for each pair of elements both operations are performed, aggregating the results step by step into a final sum. However, in general it is difficult to parallelise such an implementation, since the operations are entangled with each other at every step of the for-loop, leading to algorithmic structures which are hard to disentangle, especially when side-effects arise. This problematic character is confirmed by the extensive research to automatically parallelise for-loops in existing code [5,16].

We will choose a different approach by looking at such algorithms from a more abstract perspective: instead of defining the dot-product by using indices and by intertwining the operations $+$ and $\cdot$ together into one computational activity, we will "lift" the operations to the aggregate level, in this case to the vector level. From Eq. (69), it can be seen that we need:

- pairwise multiplication of the elements of the vectors. We use the notation $\widehat{\cdot}$ for this lifted version of multiplication. Note that this usage of the operator $\widehat{\ }$ is in accordance with earlier usage, since a vector can be seen as a function from indexes to values.

- the reduction of the resulting values to a single value by using $+$. We use $\oplus$ to denote this interpretation of addition, i.e. the expression $\oplus\ \vec{x}$ means that the elements of vector $\vec{x}$ are summed.

We remark that this can be generalised to other operations than multiplication and addition as well.

Clearly, the dot product of two vectors $\vec{a}$ and $\vec{b}$ can now be defined as follows:

$$\vec{a} \cdot \vec{b} = \oplus \left(\vec{a} \mathbin{\widehat{\cdot}} \vec{b}\right) \tag{70}$$

Note that our notation does not involve reference to the individual elements in the vectors, so no indices are needed. What is further important to observe is that the operations $\widehat{\cdot}$ and $\oplus$ are now visible on aggregate level. In the algorithm for the dot product these operations are *separated*.

Now, it is possible to use such definitions at the aggregate level for partitioning, so that we can formalise it as a model transformation. The $\widehat{\cdot}$ operates on the data independently, so it is easy to parallelise. However the reduction operation $\oplus$ must be associative to be able to use parallelism. For example, splitting vector $\vec{a}$ and $\vec{b}$ in three sub-vectors $\vec{a}_1, \vec{a}_2, \vec{a}_3$, respectively $\vec{b}_1, \vec{b}_2, \vec{b}_3$ (where $\vec{a}$ and $\vec{b}$ are equally long) leads to the following parallelisation of the dot product:

$$\begin{aligned} e_j &= \oplus \left(\vec{a}_j \mathbin{\widehat{\cdot}} \vec{b}_j\right), \quad \text{where} \quad j = 1, 2, 3 \\ \vec{a} \cdot \vec{b} &= \oplus\ [e_1, e_2, e_3] \end{aligned} \tag{71}$$

Note that the indices here indicate that we have chosen to partition the dot product in three parts, i.e. they are part of the partitioning, not of the application definition. Further, the calculation of e_j and $\cdot$ can be pipelined and has a tree-like computational structure.

Thus, data parallelism is provided by defining the operation on aggregate data and control parallelism is provided by separating the $+$ and $\cdot$ operations and by staging the reduction operation in a tree. This last approach is an example of a divide-and-conquer strategy [28]. Next, we will present the model transformation to perform this partitioning automatically.

7.3.4 Transformation

In the previous section it has become clear that how the functionality is specified influences how much parallelism can be exploited. It is ongoing research how to transform such structures to the aggregate level automatically, and for now this is a manual process. However, when the algorithm is specified on the aggregate level, we can automatically partition it to execute data-parallel or with a divide-and-conquer strategy. This is done with a higher-order function, that takes the aggregate operation and generates a number of connected dataflow processes, i.e. the step from Eq. (70) to Eq. (71) is automated (assuming the reduction operation is associative). The granularity (the vector is split in three in our example above) is a manually specified parameter.

The amount of computation and communication per process must be matched with the capabilities of the processors and the network.

Distribution As an example we define the higher-order transformation for the implementation of the dot-product of Eq. (71):

$$\vec{h} \cdot \vec{x} = \oplus \; (\vec{h} \mathbin{\hat{\cdot}} \vec{x}) \tag{72}$$

To distribute the dot-product the inputs are split every n values. The function $split$ cuts a vector $\vec{x}$ in a sequence of sub-vectors of length n:

$$split_n \; (\vec{x}) = [\vec{x}_1, \vec{x}_2, \ldots] = \mathbf{x} \tag{73}$$

It can be defined recursively as follows:

$$split_n \; ([\,]) = [\,] \tag{74}$$

$$split_n \; (\vec{x}) = \vec{a} : split_n \; (\vec{b}) \tag{75}$$

where

$$(\vec{a}, \vec{b}) = splitAt_n \; (\vec{x}) \tag{76}$$

Herein, $splitAt_n$ splits the vector $\vec{x}$ into a vector of the first n elements ($\vec{a}$) and a vector of the remaining elements ($\vec{b}$).

Furthermore, we normalise the coefficients $\vec{h}$ (h is the first element of $\vec{h}$, and $\vec{h}'$ the remaining part):

$$normalise \; (\vec{h}) = 1 : \widehat{(/h)} \; (\vec{h}') \tag{77}$$

which is only allowed if the function we distribute, the dot-product in our case, is distributive over addition.

Finally we define a generic distribution transformation for any reduction function f that takes two arguments ($\vec{h}$ and $\vec{x}$) and is distributive. For a vector $\vec{x}$ smaller than n we have:

$$distribute_{(n,f)} \; (\vec{h}, \vec{x}) = f \; (\vec{h}, \; \vec{x}) \tag{78}$$

whereas for arbitrary larger vectors $\vec{x}$ we have

$$distribute_{(n,f)} \; (\vec{h}, \vec{x}) = distribute_{(n,f)} \; (\vec{h}', \; \vec{x}\,') \tag{79}$$

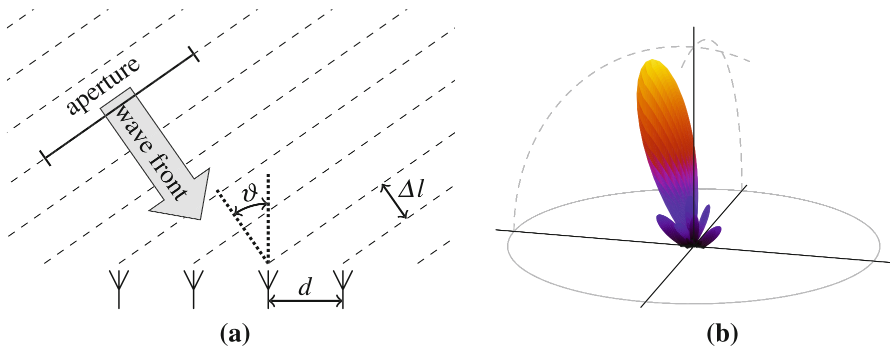

Fig. 12 Wavefront arriving at the antennas of a phased array system and its corresponding radiation pattern. (**a**) Received wavefront, (**b**) 3D radiation pattern

where

$$\mathbf{x} = split_n\ (\vec{x}) \qquad \mathbf{h} = split_n\ (\vec{h}) \tag{80}$$

$$\mathbf{h}' = \widehat{normalise}\ (\mathbf{h}) \qquad \vec{h}' = \widehat{head}\ (\mathbf{h}) \tag{81}$$

$$\vec{x}\,' = \widehat{f}\ (\mathbf{h}', \mathbf{x}) \tag{82}$$

Here the inputs are split every n values, the coefficients are normalised and the results are recursively distributed again. Only the granularity n and the reduction function f need to be specified. Note that the bold letters indicate vectors of vectors.

8 Case Study

A phased array antenna receiver system is used as a non-trivial case-study. The case contains mixed-signal aspects and signal processing on a tiled architecture.

8.1 Specification

Phased array beamforming systems use multiple antennas in an array to make a directional receiver; the directional sensitivity of the array is shown in the radiation patterns in Fig. 12b. A direction of maximum sensitivity is called a beam because of its shape. Using beam-control processing the shape and direction of the formed beam can be controlled electronically, called beamsteering (BS), or the direction of arrival (DoA) of a received signal can by estimated.

Beamforming is based on interference. A wavefront arrives at different times at the antennas because of the path length difference (see Fig. 12a). For an antenna distance d and a wavefront angle ϑ, the delay between arrival times is $\Delta t = \frac{d \cdot \sin(\vartheta)}{c}$ (c is the propagation speed of the radio waves). Depending on the frequency of the wave, this time delay results in a phase shift ($\Delta\psi = \omega \cdot \Delta t$) giving rise to the term "phased array". After reception at the antenna elements, the radio frequency (RF) front-end

performs down-conversion, followed by antenna processing (AP) for calibration and equalisation purposes. A beamformer adds the signals from the antennas together. They add up constructively if they are in phase. By correcting the delay for a certain angle we can steer the direction of maximum sensitivity.

The specification is defined as follows. As described in Sect. 3.1 we consider the *delay* of a signal s with a value $\delta = \Delta t$ as a *signal transformation*, i.e, as a function with signal s as argument and a changed signal as result. Remember that this delayed signal is defined as

$$delay_\delta\,(s) = t \mapsto s\,(t - \delta)$$

Thus, the sequence of values of the delayed signals at time t for the array of antennas $\langle a_1, a_2, \ldots\rangle$ is (s is the original signal):

$$\sigma(t) = \langle delay_{\delta_1}(s)(t), delay_{\delta_2}(s)(t), \ldots\rangle$$

The phase correction w_i for antenna a_i and steering direction $\langle\alpha_0, \gamma_0\rangle$ is calculated as (λ is the wave length of the carrier):

$$w_i = e^{j \cdot \frac{2\pi}{\lambda} \cdot \Delta l_i}$$

where Δl_i is the difference in length between the origin and antenna a_i, projected in the steering direction, and calculated as follows ($\cdot$ is the dotproduct of two vectors):

$$\Delta l_i = \langle x_i, y_i, z_i\rangle \cdot \langle 1, \alpha_0, \gamma_0\rangle$$

Let $\vec{w}$ be the steering vector $\langle w_1, w_2, \ldots\rangle$ and let $\vec{w}^*$ be its complex conjugate. Then the *beamformer bf* applies a phase shift correction and is defined as the function (N is the number of antennas):

$$bf_{\vec{w},\sigma} = t \mapsto \frac{\vec{w}^* \cdot \sigma(t)}{N}$$

These definitions are directly implemented in Haskell.

8.2 Co-design

The specification of the simple beamforming system is refined with structural details illustrated by the block diagram of Fig. 13. At the top level of the design the model consists of the environment followed by the beamforming system:

$$model = environment \triangleright system$$

Below we will explain how the environment and the system are formalised, and how the model is simulated over a sequence of time steps.

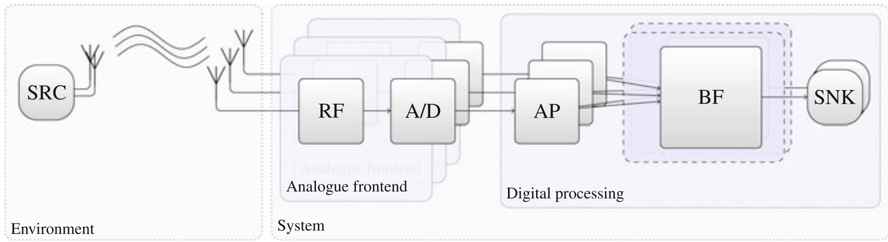

Fig. 13 UNITI simple beamformer model

8.2.1 Environment

First of all, the *environment* models one or more sources together with their transmitters that send the signals. We will assume that a source generates the signal as it is sent by a transmitter. Suppose

$$sources \;=\; (src_1, src_2, \ldots)$$

that is

$$sources \;=\; src_1 \parallel src_2 \parallel \ldots$$

The fact that a source *generates* a signal formally means that each src_j is a function such that $src_j()$ is a *signal*. Each antenna receives the signal from each source through a *channel* for which we now model the time delay. A channel ch_{ji} from source j to antenna i in fact is a *signal transformation* which delays the signal s_j:

$$ch_{ji}(s_j) \;=\; delay_{\delta_{ji}}(s_j)$$

Since the DoA d_j of source j and the position p_i of antenna i are known, δ_{ji} can be calculated as in Sect. 8.1.

The *environment* now is the total of the sequential compositions for all j, i of src_j and ch_{ji}:

$$env_{ji} \;=\; src_j \triangleright ch_{ji}$$

Let

$$chs_i \;=\; ch_{1i} \parallel ch_{2i} \parallel \ldots$$

then

$$sources \triangleright chs_i$$

delivers all signals for antenna i. Now let

$$channels \;=\; chs_1 \parallel chs_2 \parallel \ldots \parallel chs_N$$

then the total environment is

$$environment \;=\; (\parallel^* \; sources) \rhd channels$$

where $\parallel^*$ creates as many copies of $sources$ (in parallel) as needed.

8.2.2 System

The intended architecture is a tiled multi-core SoC for the beam-control and beam-former processing software. The RF front-end is implemented in analogue hardware as the frequencies are too high to allow the use of digital hardware. After down-conversion the signals are digitised and filtered by the AP block in fixed digital hardware. Thus, between the RF frond-end and the AP for each antenna, an ADC is added. The DF domain is used to model the beamformer application running on the multi-core architecture. For brevity the beam-control processing is left out. See [29] for more about beam-control processing on a multi-core platform.

The $system$ consists of (i) the parallel composition of a $frontend$ for each antenna, (ii) a part which $processes$ the outputs from these antenna frontends and produces beams, and (iii) the parallel composition of a sink (snk) for each beam to plot the result, i.e.

$$system = (\parallel^* \; frontend) \rhd processing \rhd (\parallel^* \; snk) \tag{83}$$

Concerning (i), the frontend of each antenna, we remark that it consists of a receiver rx, an rf frontend and an adc, i.e.

$$frontend \;=\; rx \rhd rf \rhd adc$$

The receiver rx gets delayed signals from each source, i.e. rx is a function which adds a sequence of delayed input signals into a single output signal as defined in Sect. 7:

$$rx = \oplus$$

Note that we consider rx here as part of the system under design, and that the signals from the environment are combined at the receiver antennas. Note also that the rf and adc for each antenna are the same. Here we will assume that the signal transformation rf only passes through the signal, i.e. rf is the identity function, though for later refinements of the model, rf can be defined differently.

The adc (with sample period h) is defined as in Sect. 3.2:

$$adc_h\,(s) \;=\; t \mapsto s(\lfloor t/h \rfloor \cdot h)$$

The frontend is identical for each antenna. That is possible because the influence of the position of the antenna is already accounted for in the delay of each input signal, as discussed above. To combine the frontends of all antennas into one component, we simply have to compose as many frontends in parallel as there are antennas.

Concerning (ii), the processing part consists of antenna processing (ap) and beamforming (bf), where ap is dealt with in the DT domain and beamforming in the DF domain. As with the rf frontend we assume that ap is the identity function, though it might be defined differently. There are as many ap components needed as there are antennas and their outputs are input for the beamforming operation.

The definition of bf differs from the definition in Sect. 8.1 where bf was a signal transformation in the CT domain, whereas now it is in the DF domain, i.e. a function from input tokens to output tokens. Thus the functionality of the beamformer process in the DF domain is:

$$bf_{\vec{w}}\ (\vec{x}) \ = \ \frac{\vec{w}^* \cdot \vec{x}}{N}$$

In order to wrap the function bf in a DF component, we first have to apply the $\boxdot$ operator and then we have to initialise the internal state with an empty state using the $\Uparrow$ operator (see Sect. 3.3).

The total processing chain now becomes:

$$processing \ = \ (\|^* \ ap) \triangleright (\boxdot \, bf_{\vec{w}} \Uparrow [\,])$$

where $\vec{w}$ is the same steering vector as defined in Sect. 8.1.

The function bf calculates a single beam. Without going into details, we mention that in case more than one beam has to be formed using the same antenna signals, we defined a composition operator $\triangleright^*$ which duplicates the input signals to match the number of beamformers. Note that duplication of input signals is not the same as parallel composition of signal transformers.

Finally, concerning (iii), the snk components plot the signal from each beam as a side-effect and returns a vacuous output.

8.2.3 Simulation

The model as derived above contains components in the CT domain ($sources$, $channels$, rx, rf, adc), in the DT domain (ap), and in the DF domain ($(\boxdot bf_{\vec{w}}) \Uparrow [\,]$). The sequential composition operator takes care that the various domains are integrated, e.g., by embedding the ap component defined in the DT domain in a CT domain component (see Sect. 4.4).

Again, all definitions above can be straightforwardly reformulated in Haskell. Since the composition operators are also defined in Haskell, the whole model can be simulated by evaluating it as a single Haskell program.

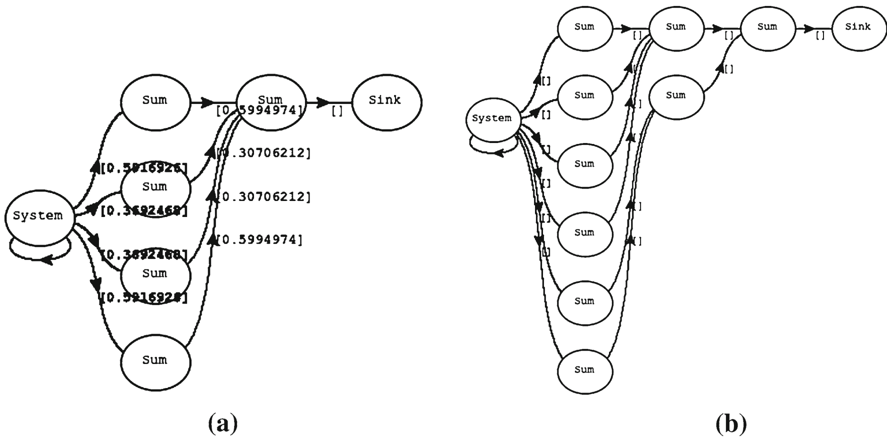

Fig. 14 Dataflow models of an automatically partitioned beamformer. (**a**) 16 antenna beamformer, (**b**) 23 antenna beamformer

8.2.4 Radiation Pattern

As an example of the flexibility of the definitions, we reuse the components from the definitions above to generate radiation patterns. This is achieved by calculating the transfer function over all angles.

The radiation pattern is calculated by setting the (complex) source signal to $1\,e^{j\,0}$ and calculate the result over all source angles (α, γ) with a fixed steering angle (α_0, γ_0):

$$P(\alpha, \gamma) = |bf_{(\alpha_0,\gamma_0)}(\vec{x})|$$

where

$$\vec{x} = (src \rhd channels \rhd (\|^* \; frontend))\,()\,(0) \tag{84}$$

$$src\ () = t \mapsto \left((\alpha, \gamma)\, , e^{j \cdot 2\pi f_c t}\right) \tag{85}$$

Note that the source *src* now is a function which yields the complex signal including its corresponding DoA (α, γ).

8.3 Partitioning

The beamformer combines all the antenna signals and is a computationally intensive block of the system, but still requires flexibility for beam-control. The beamformer is therefore partitioned to be mapped to a multi-architecture.

Beamforming is defined as the dot-product of the antenna signals with a correction vector (see above). By defining beamforming as an aggregate operation, the reduction operation is recognised and the model transformation from Sect. 7 is used to partition the beamformer. A nice property of this model transformation is that each part in the

partitioning performs the same functionality. The transformation on the beamformer is then defined as:

$$distribute_n \ (bf\ (\vec{w}, \vec{x})) \tag{86}$$

with the definition of $distribute$ from Sect. 7. Note that this definition is independent of the number of antenna elements.

The intended core is a 200 MHz VLIW processor with 5 ALUs. We find that we can beamform 4 antenna signals per core, because a complex multiplication needs 4 multipliers and the data-rate per antenna is 50 MS/s. The next step is to explore the design space of the number of antennas the system can support. For 16 antennas this results in a hierarchical beamformer with 5 processes (shown in Fig. 14a) and thus 5 cores. For 23 antennas the beamformer consists of 9 processes, shown in Fig. 14b which matches a 3×3 grid of cores well, leaving some processing capacity for the beam-control calculations. Both partitionings are generated automatically, only the number of the antennas and their positions are changed.

9 Results

The main evaluation criteria for UniTi are the effectiveness and usefulness of the approach. As such criteria are difficult to quantify objectively, we will present the applicability and flexibility of UniTi for the phased array beamforming case study, in order to provide an indication.

9.1 Applicability

See Fig. 15 for a screenshot of the complete framework during the simulation of the beamforming case study with a 5×5 array, a 30° steering direction and two sources, one of which is filtered away. It shows the results of a simulation of the CT, DT and DF domains in a single model, including structural hierarchy in the system overview and the processes during execution of the DF model. The model is executed for simulation, allowing step by step evaluation of the behaviour of the system. Additionally a radiation pattern shows the current steering direction of the beamcontroller.

9.1.1 Performance

Comparison of the case study (without the dataflow model) with an implementation in Simulink gives ∼3 times speed-up in our advantage [21]. This is because the time step used by the solver in Simulink must be reduced substantially to achieve enough accuracy for the time delays at the channel, resulting in many more evaluations of the model than for the UniTi implementation.

Profiling results for an increasing number of antennas (3×3, 5×5 and 11×11 arrays) are shown in Fig. 16. The top two graphs show the results of only the CT and DT version of the model (with the beamformer in the DT domain). The next two graphs include the radiation pattern, which is computationally expensive. The final

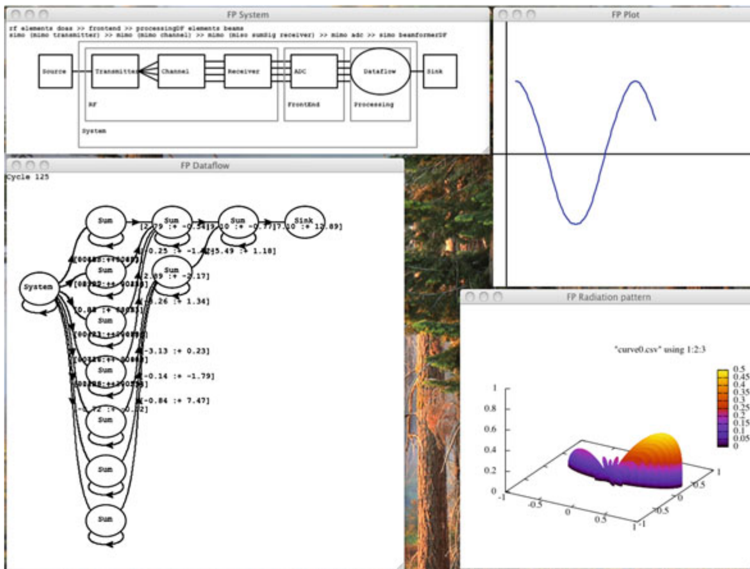

Fig. 15 Screenshot of the UNITI framework

two also include the dataflow model (by lifting the beamformer function to a DF component), including the visualisation of the DF processes. We can see that the memory requirements grow faster with more antennas than the processing requirements. This limits the simulation to a few hundred antennas for a 2 GHz Core2 Duo with 4 GB RAM. The Simulink version, however, is limited to even fewer antennas (taking 236 s for an 11×11 array without radiation pattern or dataflow model). Further, the radiation pattern calculation becomes dominating with larger arrays. Though, because the radiation pattern is only for illustration, it can be approximated with a function instead of being calculated over all angles, or it can be turned off completely. Note that the instantiation of the wxWidgets toolkit for the graphical user interface (GUI) has a relatively large but fixed processing overhead compared to the model. With the dataflow model included the GUI is also dependent on the array size, because a larger array results in a larger number of dataflow processes to be visualised. Note also that the beamforming processing, which was a computationally expensive part of the semantic model, is now performed by the dataflow model. The dataflow implementation is more strict, taking more time but requiring less memory.

As the framework was developed as a proof of concept, we expect there is ample room for improvement in efficiency, especialy for reducing the memory requirements enabling the simulation of larger arrays.

9.1.2 Designer Productivity

Much of the design flow is automated by the framework, i.e. multi-domain composition, communication and synchronisation for a DF model and model transformations. Table 1 shows the code size in lines. We see that the framework is 2,500 lines, half

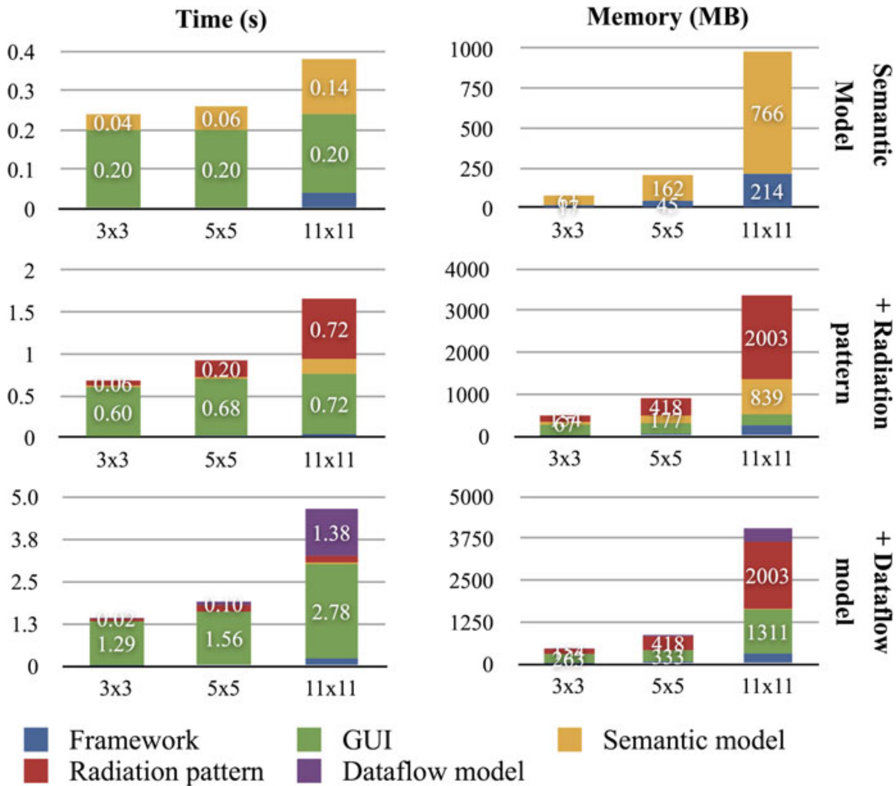

Fig. 16 Profiling results (time (s) and memory (MB)) for the simple beamformer case study on a 2GHz Core 2 Duo with 4GB RAM

of which is of the dataflow support. The case is about 600 lines, with the majority in the system model as this implements the functionality of the system and is re-used for the dataflow model. A large part of the code (the framework) is thus re-useable, providing the glue-logic. The additional code needed in the model because of the design framework is very little, about 5 %.

It is difficult to estimate and compare the development time of the case study with a Simulink implementation, because the UniTi version was used to develop the frame-

Table 1 Code size (lines)

	Framework	Case
Support	811	
GUI	411	
System model	141	438
Radiation pattern	10	46
Dataflow model	1143	135
Total	2516	619

work. The graphical representation of Simulink is more intuitive when developing the initial model. However, with equal knowledge of the tools, we expect the UniTi approach to be more productive because the higher abstraction level of the implementation improves flexibility (see below) making adjustments easier. Furthermore, changes are checked by the type system and transformations are defined to be correctness preserving.

9.1.3 Flexibility

The presented design flow of UniTi and the guidelines for specifying the algorithm using aggregate operations aim at increasing the flexibility. For example, the number of sources or antenna elements and their positions can be changed without the need to change the model, and higher-order model transformations are used for automated partitioning. This enables us to quickly evaluate design alternatives.

Automation Composition, simulation and multi-domain integration are automatically provided by the framework. Implementing the functionality is of course manual. Design decisions for dividing functionality over domains and specifying the algorithm so it can be partitioned effectively are also the designer's responsibility. Still, lifting functions to operate on multiple elements, and partitioning using data and control parallelism with such aggregate structures is automated.

Scalability Specifying the algorithm at a higher abstraction level makes it independent from the number of elements and enables automated model transformations, thereby improving the scalability of the design. The framework itself (for our case) scales linearly in performance with the number of antennas as shown in Fig. 16.

10 Conclusion

In this paper we have presented UniTi to address the shortcomings in current multi-domain modelling and simulation tools. UniTi provides a unified perspective on composition (using components and signals) and time, yet distinguishes between different domains and different notions of time. For this unified representation, DF signals represent token updates to channels, and DF components represent processes together with its input channels and firing rules. By generalising this notion of components and signals, we can integrate components in the DF domain with components in the CT and DT domain. As a result, UniTi allows for multi-domain simulation in a single model, including exact time transformations, different time granularities and local solvers. Furthermore, UniTi provides model transformations, which transform (parts of) the model to a more refined model with correctness preserving mathematically based transformations. UniTi combines all this in a design flow and framework for the design of embedded systems, which is illustrated by a case study on a phased array beamforming system design.

Acknowledgments This research is partly funded by STW project NEST (10346) and EU-FP7 project S(o)OS (248465).

References

1. Carloni, L.P., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.L.: Languages and tools for hybrid systems design. Found. Trends Electron. Des. Autom. **1**(1/2), 1–193 (2006). doi:10.1561/1000000001
2. Courtney, A., Elliott, C.: Genuinely functional user interfaces. In: ACM SIGPLAN Haskell Workshop (HW'2001), pp. 41–69 (2001)
3. Eker, J. et al.: Taming heterogeneity—the Ptolemy approach. Proc. IEEE **91**(1), 127–144 (2003). doi:10.1109/JPROC.2002.805829
4. Elliott, C., Hudak, P.: Functional reactive animation. In: 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP '97), pp. 263–273. ACM (1997). doi:10.1145/258948.258973
5. Erbas, C., Pimentel, A.D., Thompson, M., Polstra, S.: A framework for system-level modeling and simulation of embedded systems architectures. EURASIP J. Embed. Syst. 2007(1) (2007). doi:10.1155/2007/82123
6. Feng,T.H., Lee, E.A.: Scalable models using model transformation. In: 1st International Workshop on Model Based Architecting and Construction of Embedded Systems (ACESMB). EECS Department, University of California, Berkeley (2008)
7. Fritzson, P., Engelson, V.: Modelica—a unified object-oriented language for system modeling and simulation. In: Jul, E. (ed.) ECOOP'98—Object-Oriented Programming. Springer, Berlin (1998). doi:10.1007/BFb0054087
8. Henzinger, T.A., Sifakis, J.: The embedded systems design challenge. In: 14th International Symposium on Formal Methods (FM 2006), pp. 1–15. Springer (2006)
9. Hudak, P., Courtney, A., Nilsson, H., Peterson, J.: Arrows, robots, and functional reactive programming. In: Advanced Functional Programming, pp. 159–187. Springer, Berlin (2003). doi:10.1007/978-3-540-44833-4_6
10. Lee, E.A.: Cyber physical systems: design challenges. In: 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC 2008), pp. 363–369. IEEE (2008). doi:10.1109/ISORC.2008.25
11. Lee, E.A.: Computing needs time. Commun. ACM **52**(5), 70–79 (2009). doi:10.1145/1506409.1506426
12. Lee, E.A., Messerchmitt, D.G.: Synchronous data flow. Proc. IEEE **75**(9), 1235–1245 (1987). doi:10.1109/PROC.1987.13876
13. Lee, E.A., Parks, T.M.: Dataflow process networks. Proc. IEEE **83**(5), 773–801 (1995). doi:10.1109/5.381846
14. Lee, E.A., Sangiovanni-Vincentelli, A.L.: A framework for comparing models of computation. IEEE Trans. Computer-Aided Des. Integr. Circuits Syst. **17**(12), 1217–1229 (1998). doi:10.1109/43.736561
15. Najjar, W.A., Lee, E.A., Gao, G.R.: Advances in the dataflow computational model. Parallel Comput. **25**(13–14), 1907–1929 (1999). doi:10.1016/S0167-8191(99)00070-8
16. Nikolov, H., et al.: Daedalus: toward composable multimedia MP-SoC design. In: 45th Annual Design Automation Conference (DAC'08), pp. 574–579. ACM (2008). doi:10.1145/1391469.1391615
17. Object Management Group, Inc. (OMG): OMG systems modeling language (OMG SysML). Technical report version 1.1 (2008)
18. OMG Architecture Board ORMSC: Model driven architecture (MDA). Technical report ormsc/2001-07-01 (2001)
19. Peterson, J., Hager, G.D., Hudak, P.: A language for declarative robotic programming. In: IEEE International Conference on Robotics and Automation, pp. 1144–1151. IEEE (1999). doi:10.1109/ROBOT.1999.772516
20. Reekie, H.J.: Realtime signal processing: dataflow, visual, and functional programming. Ph.D. thesis, University of Technology Sydney (1995)
21. Rovers, K.C.: Functional model-based design of embedded systems with UniTi. Ph.D. thesis, University of Twente (2011). doi:10.3990/1.9789036532945
22. Rovers, K.C., van de Burgwal, M.D., Kuper, J., Kokkeler, A.B.J., Smit, G.J.M.: Multi-domain transformational design flow for embedded systems. In: International Conference on Embedded Computer

Systems (SAMOS 2011), pp. 93–101. IEEE Computer Society (2011). doi:10.1109/SAMOS.2011.6045449
23. Rovers, K.C., Kuper, J., van de Burgwal, M.D., Kokkeler, A.B.J., Smit, G.J.M.: Mixed continuous/discrete time modelling with exact time adjustments. In: 7th International Wireless Communications and Mobile Computing Conference (CyPhy'11), pp. 1111–1116. IEEE (2011). doi:10.1109/IWCMC.2011.5982696
24. Rovers, K.C., Kuper, J., Smit, G.J.M.: The problem with time in mixed continuous/discrete time modelling. ACM SIGBED Rev. **8**(2), 27–30 (2011). doi:10.1145/2000367.2000373
25. Sander, I.: System modeling and design refinement in ForSyDe. Ph.D. thesis, KTH Royal Institute of Technology (2003)
26. Sander, I., Jantsch, A.: System modeling and transformational design refinement in ForSyDe. IEEE Trans. Computer-Aided Des. Integr. Circuits Syst. **23**(1), 17–32 (2004). doi:10.1109/TCAD.2003.819898
27. Soliman, S.S., Srinath, M.D.: Continuous and Discrete Signals and Systems, 2nd edn. Prentice Hall, Englewood Cliffs, NJ (1998)
28. Trinder, P.W., Hammond, K., Loidl, H.W., Peyton Jones, S.L.: Algorithm + strategy = parallelism. J. Funct. Program. **8**(1), 23–60 (1998). doi:10.1017/S0956796897002967
29. van de Burgwal, M.D., Rovers, K.C., Blom, K.C.H., Kokkeler, A.B.J., Smit, G.J.M.: Mobile satellite reception with a virtual satellite dish based on a reconfigurable multi-processor architecture. Microprocess. Microsyst. 1–29 (2011). doi:10.1016/j.micpro.2011.08.005
30. Vachoux, A., Grimm, C., Einwich, K.: SystemC-AMS requirements, design objectives and rationale. In: Design, Automation and Test in Europe Conference and Exhibition (DATE 2003), pp. 388–393. IEEE (2003). doi:10.1109/DATE.2003.1253639
31. Wan, Z., Taha, W., Hudak, P.: Real-time FRP. In: 6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01), pp. 146–156. ACM (2001). doi:10.1145/507635.507654
32. Wiggers, M.H.: Aperiodic multiprocessor scheduling for real-time stream processing applications. Ph.D. thesis, University of Twente (2009). doi:10.3990/1.9789036528504
33. Zheng, H.: Operational semantics of hybrid systems. Ph.D. thesis, University of California Berkeley (2007)