

Mixed continuous/discrete time modelling with exact time adjustments

Kenneth C. Rovers, Jan Kuper, Marcel D. van de Burgwal, André B.J. Kokkeler and Gerard J.M. Smit

Computer Architecture for Embedded Systems group

CTIT, Department of EEMCS, University of Twente

P.O. Box 217, 7500 AE Enschede, The Netherlands

{K.C.Rovers, J.Kuper, M.D.vandeBurgwal, A.B.J.Kokkeler, G.J.M.Smit}@utwente.nl

Abstract—Many systems interact with their physical environment. Design of such systems need a modelling and simulation tool which can deal with both the continuous and discrete aspects. However, most current tools are not adequately able to do so, as they implement both continuous and discrete time signals as consisting of separate values at a single global simulation clock. The consequence is that simulation, of a time delay for example, either yields inaccurate results or becomes inefficient.

We propose a solution by considering (continuous) signals as functions of time and by separating different notions of time. Signals thus correspond directly to their mathematical representation and e.g. time delays can be dealt with exactly. A second advantage is that discretisation of time can be dealt with locally, such that numerical approximations in the continuous time domain or sampling of the ADC can be calculated without influencing the time granularity of the rest of the system.

To handle such signals, we need higher order functions. As they are standard in functional languages, we implement our approach in Haskell. We illustrate the approach with a case study on beamforming in phased array systems.

Index Terms—heterogeneous modelling, continuous time, discrete time, simulation, time delay, beamforming

I. INTRODUCTION

The importance of modelling multiple (time) domains when designing systems is increasing. In particular, in designing so-called *cyber-physical systems*, i.e. embedded systems which include peripherals interfacing with the physical world, both *continuous time* (CT) and *discrete time* (DT) have to be taken into account [1]. For example, in a radio receiver the CT processes include the actual radio signal together with the distortions introduced by a channel and the analogue front-end of the receiver, whereas the further processing of signals is done in the DT domain. During analogue/digital co-design of such a *mixed-signal system*, we need an integrated model which supports both simulation and verification as well as iterative development (model-based design).

Current design tools such as Simulink and Ptolemy allow modelling multiple domains. For simulation purposes these (and other) tools choose one single (global) time step on which the whole system under design is evaluated (see [2] for an extensive survey of multi-domain system design tools). This time step should be small enough to meet the requirements of *all* the components in the system. However, this may result in

extremely inefficient simulation as several components such as integration can require a very small time step to achieve enough accuracy. Secondly, it will not be possible to capture all timing issues in a global clock. For example, a time delay may depend on changing circumstances in the real world and thus will not be predictable during simulation. Since existing simulation tools also consider CT as a sequence of discrete values, the best they can offer for the value of a delayed signal is an interpolation between known values on moments that are close to the delay. However, modelling the physical signal should be *exact* for correctly validating the design, while interpolation introduces non-existing inaccuracies. For example, testing algorithms for wideband beamforming or time shifted sampling relies on this.

Summarising, both aspects — using values for continuous functions and a single global time step for simulation — cause that simulation either becomes less accurate or less efficient.

In this paper we propose a novel way to simulate CT/DT systems which is more accurate while retaining efficiency. 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 piecewise horizontal from the last sample of the ADCs. Thus, our simulation technique coincides with the standard mathematical modelling of such systems. Additionally, in our approach time is kept local, i.e. every continuous component may have its own discretization 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.

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.

More specifically, in this paper we will:

- analyse the problem (section II)
- present mixed-signal modelling with signals as functions of time (and/or space) and parts of the system as blocks (components) that transform these signals (section III),
- provide an implementation in a functional language, thereby staying close to the mathematical description,
- show how to compose components (section IV),
- and provide an extensive case study in the form of a beamforming application (section V).

This research is partly funded by Thales Nederland B.V. and STW projects CMOS Beamforming (07620) and NEST (10346).

II. ANALYSIS

Consider a simple system consisting of a continuous time sine source connected to a time delay block, followed by an ADC, a bias (offset) and a sink which plots the output. This is shown in figure 1. The CT part and DT part are indicated in the figure.

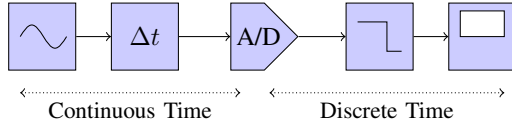


Fig. 1. Simple system block diagram

A. Time

In such systems we identify different notions of 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 locally, possibly transformed by e.g. a time delay, the *local time*,
- the time steps that are necessary to numerically approximate functions (e.g. an integral), the *approximation time*.

The third notion of time is necessary to represent relativity: different distances from a source lead to different local time references to the source. This occurs for example for a front-end with multiple (differential) signal paths, which might have slightly different path lengths, thereby modelling non-ideal common mode noise rejection.

B. Signals

Furthermore, we identify two kinds of signals which represent the connections between the system components:

- signals in the CT domain are functions of time or functions of space and time, i.e. they represent the value of the system over all time,
- signals in the DT domain are piecewise horizontal, i.e. they represent the latest value at the sample time for the current local time.

C. Simulation

Existing simulation tools do not distinguish these different notions of time. All the notions of time are coalesced into the simulation time. Furthermore, most existing tools have a single representation of signals as values at a certain simulation time.

Simulating an even simpler system in Simulink (shown in figure 2) illustrates the problem. This system consists of a source, a time delay and a sink. A 1Hz sine source with a 0° initial phase is used. The simulation time step is 0.04s, so 25 simulation results per period. The time delay is 0.1s, i.e. the local time is shifted by 0.1s. Note that this is not an integral multiple of the step size. The scope output is shown in figure 3. The result resembles a shifted sine wave perfectly well. However, figure 4 shows the error when compared with an ideal result (a sine wave with an initial phase of $-2 \cdot \pi \cdot 0.1$).

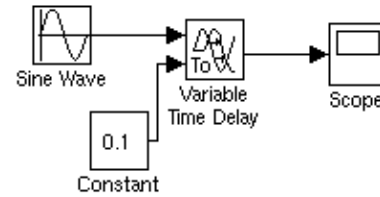


Fig. 2. Simulink example

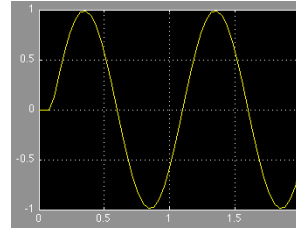


Fig. 3. Delayed sine wave

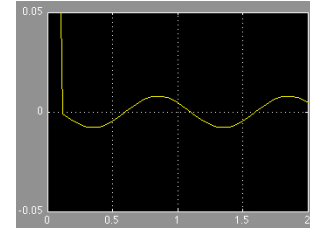


Fig. 4. Time delay error

The error is caused by interpolation. Samples at simulation time are buffered by the time delay block. When a sample at the delayed time is not available, the result is interpolated from the surrounding samples. If the simulation time steps are large compared to the frequency, this error can be quite substantial.

A natural solution would be to choose the simulation time steps in correspondence with the time delay. The time delay block buffers *values* in order to retrieve the value at the delayed time. For an accurate result the simulation time step must thus be a *multiple of the delay*. However, this is a problem when:

- the delay is small, as this results in many (fixed) steps,
- there are multiple time delays without common factors as this forces separate simulation times for each time delay,
- the delay is variable, as we would need to know the delay in the future.

As in general the time delay *can* be variable, in Simulink the delay is not even taken into account when determining the time step. The consequence is that if a value is not available at the exact time, it is interpolated between available values, giving results that are not *exact*.

The ADC *is* assumed to have a fixed sample rate and thus fixed step size. Implemented as a zero-order-hold (ZOH), the simulation times of the system *are* synchronised with the sample times for both fixed and variable step sizes in Simulink. Adding a second ADC with a different sample rate results in a multi-rate system. Such a system has its sample times matching with the sample rates of one of the ADCs for variable step sizes. However, for fixed step sizes, the simulation tool is forced to generate a step size small enough to approximate a common factor. For Simulink, with more than two ADCs this causes an error because the step size becomes too small.

The time steps needed for numerical approximation of, e.g. an integral, often have to be very small for sufficient accuracy. The global clock then causes that the whole system is simulated at this fine-grained time step, leading to severe inefficiencies. For example, the simulation of one second of a phased array antenna system took tens of minutes in Simulink and minutes when precompiled [3].

III. BASIC CONCEPT

To support accurate simulation at reasonable performance without the identified problems, we must separate the notions of time and allow components to change the time reference.

A. Mathematical model

Looking at the mathematical representation of a system, it does provide such an abstraction. The standard mathematical interpretation of a signal flow diagram is that the arrows express *signal functions* and the components denote *signal transformations*, i.e. they transform signal functions. Furthermore, the diagram as a whole then is a *composition* of these transformations. The diagram in figure 1 can be expressed as follows (the bias is represented as adding 1):

$$source \triangleright delay_{0.2} \triangleright adc_{0.3} \triangleright add_1 \triangleright sink$$

where \triangleright is the operation to compose transformations. Let φ, ψ be transformations, i.e. components; the notation $x \mapsto ..x..$ denotes the function which maps x to $..x..$:

$$\varphi \triangleright \psi = f \mapsto \psi(\varphi(f))$$

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.

In figure 1 the first component *source* generates a source function, say f , which may express some combination of trigonometric functions. Note that we consider *source* as a transformation as well, i.e. it needs a vacuous argument to deliver a signal function f :

$$source () = f$$

The next component, *delay*_{0.2}, denotes a function which *delays* the signal with a 0.2 time units. This time delay takes place in the CT domain, so it may have any value or even a variable value. As mentioned, for current simulation environments it is a problem to model this time delay exact, since in these environments also the CT domain is discretized for simulation. Mathematically, however, the time delay transformation is simply defined as:

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

Thus, to find the function value on time t *after* a 0.2 time delay, one has to know the function value at time $t - 0.2$.

The next component is the ADC which transforms the signal f into a discretized signal with time interval d (0.3). The effect of *adc* _{d} can be defined as a piecewise horizontal function:

$$adc_d(f) = t \mapsto f(\lfloor t/d \rfloor \cdot d)$$

Applying this result to a time t gives the latest value that was sampled at or before t .

The transformation for *add* _{x} that adds x to all values of the signal function is fairly trivial:

$$add_x(f) = t \mapsto x + f(t)$$

Finally, the component *sink* makes a plot of the resulting function, hence, *sink* as a *transformation* may be rather meaningless. Here we assume that the values of the signal after plotting are indeed thrown away, i.e. *sink* is defined as:

$$sink (f) = ()$$

Now suppose

$$source () = sin$$

then the result of (the interesting part of) the signal flow diagram in figure 1 is:

$$\begin{aligned} & (delay_{0.2} \triangleright adc_{0.3} \triangleright add_1) sin \\ & = add_1 (adc_{0.3} (delay_{0.2} sin)) \\ & = t \mapsto 1 + sin(\lfloor (t - 0.2)/0.3 \rfloor * 0.3) \end{aligned}$$

Note that the time delays are accurately included in the final discretized function.

B. Implementation

In the previous section we gave the mathematical interpretation of a simple signal flow diagram. This interpretation now has to be represented in a programming model such that the system can be evaluated for simulation purposes. A functional programming language contains abstraction mechanisms to express the mathematical interpretation in a straightforward way, such that the essential features are kept. We choose for the functional programming language Haskell.

First of all, we have to define the composition operator \triangleright in Haskell notation:

```
phi >>> psi = \f -> psi (phi f)
```

Thus, in Haskell the definition for composition is exactly the same as in mathematics (except that the argument f is not written between brackets).

Now the full system can be written as follows:

```
system = source >>> delay 0.2 >>> adc 0.3 >>> add 1 >>> sink
```

Also the transformations can be defined in the same way, for example (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 x f = \t -> x + (f t)
```

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 at the `system` definition, while the second (f) follows later when the system is evaluated. This is called *partial application*.

Now evaluating the same part of the full system as above leads to the desired result:

```
(delay 0.2 >>> adc 0.3 >>> add 1) sin
= add 1 (adc 0.3 (delay 0.2 sin))
= \t -> 1 + (sin (floor ((t-0.2)/0.3) * 0.3))
```

The important issue in this example is that the values of the function `sin` are now calculated at the delayed sample time without buffering or interpolation.

Haskell can directly express a transformation, because it has *higher order functions*, i.e. functions which may have functions as arguments and/or functions as results. Hence, it is possible to transfer signal functions between CT components, avoiding the necessity to discretize functions to values.

Simulation is performed by evaluating the system for a list of simulation times (`sink` expects a list of time instants):

```
ts = [0,0.1..1]
simulation = system ts
```

The resulting plot is shown in figure 5.

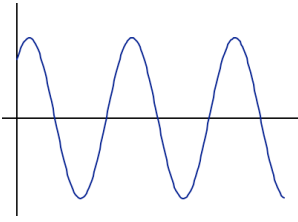


Fig. 5. Simple system simulation

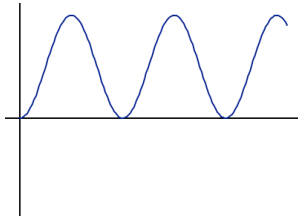


Fig. 6. Integration simulation

IV. DETAILS

Signal flow/block diagrams are popular in system design tools because of their intuitive use and easy of understanding [2]. We would like the Haskell representation to be similar.

A. Composition

Composition of functions is analogous to connecting blocks in a signal flow diagram. “Normal” functions operate on values, but signal transformations operate on signal functions. We can transform a normal function so that it operates on signals, called *lifting*, and use it for composition.

Unary operations such as $\sqrt{\quad}$ or $+1$ change the signal independent of the time:

$$\sqrt{f} = t \mapsto \sqrt{f(t)}$$

Implementation is straightforward:

```
sqrt f = \t -> sqrt (f t)
```

In fact we can introduce a function $\hat{\quad}$ to lift all unary functions (g) so that they are applied independent of time:

$$\hat{g} = f \mapsto g(f(t))$$

Binary operations such as $+$ and $*$ are similar; we evaluate both inputs at time t and then apply the operator:

$$f + g = t \mapsto f(t) + g(t)$$

To achieve a representation more like a block diagram we define a $||$ operator which evaluates two components in parallel (used nested for more than two in parallel), and a \Rightarrow operator to combine two signals with the following binary operator:

```
phi || psi = \ (f, g) -> (phi f, psi g)
phi => psi = phi >>> \ (f, g) -> psi f g
```

To illustrate their use, consider the following system:

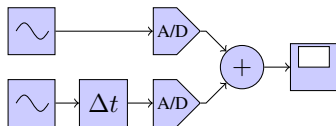


Fig. 7. Simple beamformer block diagram

This simple beamformer (section V) consists of two sources with different channel delays, which are sampled and added.

The implementation of the system in figure 7 is:

```
system = (f1 || f2) => (+) >>> sink
  where
    f1 = source1 >>> adc
    f2 = source2 >>> delay 0.2 >>> adc
    source1 () = sine f1 a1
    source2 () = sine f2 a2
```

By using a *where* clause, we introduced hierarchy, i.e. first we define two blocks to be in parallel ($f1$ and $f2$) and later we define what these blocks are. So structural hierarchy is easily achieved by naming subsystems. This possible because a composition is itself a signal transformation.

B. Calculus

Until now we have only discussed algebraic composition, for which the interesting simulation times are *at* the sample times of the ADC. Calculus is about change *over* time. For example, the voltage over a capacitor is proportional to the integration of the current through the capacitor until that time. We can solve the integral (or a differential) symbolically or numerically. A problem with symbolic integration is that for many functions a closed form does not exist. Thus, simulation tools solve the general case with numerical integration. Haskell has good possibilities to define analytic solutions to integral definitions whenever possible, but we will use numerical integration for generality.

The component for integration is defined as:

$$\int f = t \mapsto \int_0^t f(t) dt$$

Numerical integration relies on a recurrence relation to approximate the integral. A simple numerical integration method is the Euler method¹:

$$y_{n+1} = y_n + h \cdot f(t_n), \quad \text{where } h = t_{n+1} - t_n$$

So for multiple steps:

$$y_t = y_{t_0} + \sum_0^{n-1} h \cdot f(t_n), \quad \text{where } n = (t - t_0)/h \quad (1)$$

Here, h is the approximation time step that is used locally. 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), as in our approach. 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. We imagine 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, locally applying the time steps, the needed resolution is automatically passed on and adapted from the output to the inputs.

An example is the following system and its implementation:

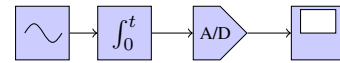


Fig. 8. Integrator

```
system = source >>> (integral h y0) >>> adc >>> sink
integral h y0 f = \t ->
  y0 + sum (map (\t -> h*(f t)) (init [0,0+h..t]))
```

with h the approximation step size and $y0$ the initial value.

Following from equation 1, the input is calculated from time 0 to t in steps of h . Each result is multiplied with h to get the approximate area and these results are summed and added to the initial value. The integration of a sine wave with 10 steps per sample period is shown in figure 6. Note that we provide the sample time t to the integral function, which then itself determines how often to calculate the input f .

¹Of 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.

C. Feedback

The integral definition includes a recurrent dependence on itself. This can also be represented with a feedback composition operator ($\gg@$) in addition to the previous sequential (\gg) and parallel (\parallel) composition operators:

```
phi >>@ psi = phi >>> loop psi
loop psi = \f -> let (g, h) = psi (f, h) in g
integral h y0 =
  id >>@ ((id || id) =>> (+) >>= (id || delay' h y0))
```

where id is the identity function and $\gg=$ duplicates the input signal. Here, one of the addition arguments is a delayed version of itself, which recurses back until the initial value.

In these implementations, the integral is recalculated from time 0 to t each simulation step. A more efficient implementation that re-uses previously calculated values, is considerably more involved and omitted because of space limitations.

D. Multi-rate

Multi-rate systems have samples generated at different rates. Such systems are typically problematic for simulation because the data must be aligned with a global clock tick. As we have separated these notions of time, this is not a problem in our approach. Consider an ADC with rate 0.8 and an ADC with rate 0.9. A simulation at time 10 just means that the ADCs output their latest samples, from time 9.6 and 9.8 respectively.

V. CASE STUDY

A more elaborate phased array beamforming case study illustrates the feasibility of a larger design, the use of structural hierarchy and the necessity of exact modelling results.

A. Phased array beamforming

Phased array systems use multiple antennas in an array to make a directional receiver (the directional sensitivity of the array is illustrated in the radiation patterns in figure 9). The beam can be pointed in a direction of interest by beamsteering.

A wavefront arrives at different times at the antennas because of the path length difference (see figure 9). 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”. 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.

We have developed a digital signal processing algorithm, to adaptively steer the beam in order to track a moving source [3], which we would like to simulate and verify. If the modelled time-delays are not *exact*, we can not differentiate if the result is distorted by the delay model or if the algorithm is not correctly steering the beam. Furthermore, a large array has many time delay blocks, causing a real problem in Simulink. In [3] we solved this by approximating the delay with a phase shift, but that is only valid for narrowband signals, excluding wideband beamforming applications.

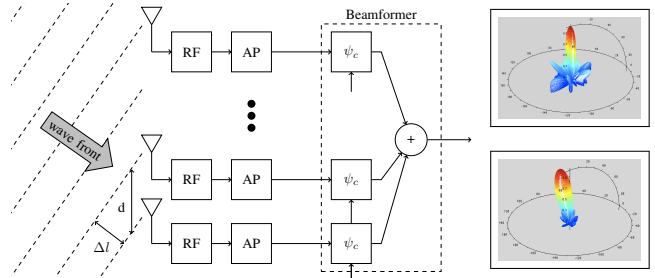


Fig. 9. Phased array receiver

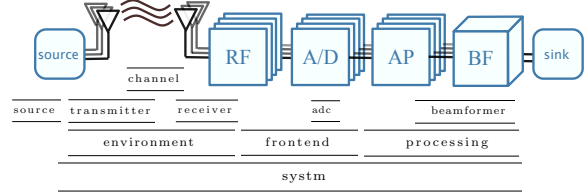


Fig. 10. Simplified system block diagram

We will use the presented approach to overcome these problems. For simplicity, we will leave out the algorithm itself and focus on the mixed-signal aspects (see figure 10).

B. System model

A signal in a beamforming system is dependent on space and time, so we use *functions of space and time* ($s \mapsto t \mapsto \dots$):

```
type Sig = S -> T -> Double
```

Furthermore, we introduce types for a direction of arrival (DOA) with range, azimuth, and elevation parameters, an element position (Pos) with cartesian coordinates, and a beam-steer direction (BSt) with two angles.

The structural hierarchy of the design is indicated in figure 10. The model supports multiple sources and multiple beams at the output. The locations of the sources (ss) and time instants (ts) are used for simulation.

```
sources = [\s t -> sine f1 t, ..]
simulation = (sources >>> system >>> sink) ss ts
```

The system is split into the environment, a receiver front-end and a digital processor (with ps the antenna element positions and bs the beam-steering directions):

```
system = environment ps >>> frontend >>> processing ps bs
```

The `environment` takes a list of sources and generates a list of antenna signals. A separate channel with a different delay is applied for each source and antenna element combination.

```
environment :: [Pos] -> [Sig] -> [Sig]
environment ps = simo transmitter ps >>> mimo channel ps
                >>> miso receiver ps
```

Therefore, `simo` (single-input-multiple-output) takes a list of sources and applies a direction dependent transmitter function for each antenna element. The `mimo` function lifts the channel function, which applies a delay for a single channel, to one that operates on these lists of lists of signals. The `miso` function combines the signals at one receiver. Although this definition is cryptic, it is completely independent on the number of antenna elements and sources, while specifying such a model graphically would be quite laborious and inflexible for more than a few antennas and sources.

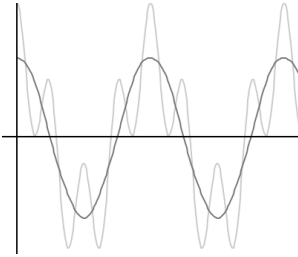


Fig. 11. Beamformer results

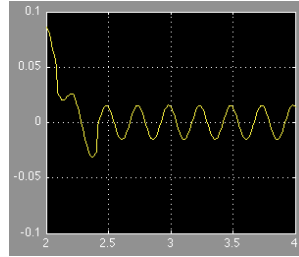


Fig. 12. Simulink error

The `channel` applies a time delay by calculating the path length between the antenna element position and the direction of arrival with the help of a coordinate transformation:

```
channel :: Pos -> Sig -> Sig
channel p f = \s t -> f s (t+dt)
  where
    dt = (delay p s)/c
    delay (Pos (x,y,z)) (DOA (r,a,e)) = lx + ly + lz
    lx = dx * x; ly = dy * y; lz = dz * z
    Pos (dx,dy,dz) = transform (DOA (r,a,e))
```

For the `frontend` we've just implemented downconversion and an ADC per antenna. The processing consists of a FIR filter (`ap`) and a `beamformer` per beam, which takes a list of antenna samples and generates an output per beam:

```
frontend :: [Sig] -> [Sig]
frontend = mimo' (down >>> adc d)
processing :: [Pos] -> [BSt] -> Sig -> Sig
processing ps bs = mimo' ap >>> simo (beamformer ps bs)
```

Figure 11 shows two sources in light-grey, a 1Hz and a 4Hz cosine. The sources have a 20° separation in azimuth angle. The result when steering the beam in the direction of the first source thereby suppressing the second source is shown in dark-grey. In the Haskell model this is exactly the result as expected, however, the same system in Simulink has an error in the range of the step size (after the start-up effect) as shown in figure 12. Both tools have a comparable execution time of around 0.93s when interpreted (2GHz Core2 Duo with 4GB). The compiled Simulink version is slower because of its overhead, while the compiled Haskell version is about 10 times faster (0.09s). The execution time scales linearly with the number of antennas.

VI. RELATED WORK

Simulink is the de facto standard for mixed CT/DT system modelling [2]. It is, however, limited in its support for multiple domains, only supporting DT as piecewise CT and Stateflow for finite state machines. This in contrast to Ptolemy [4], which provides a framework for system simulation and focusses on experimenting with various domains. SystemC is a set of C++ classes to provide discrete event simulation aimed at system-level modelling. SystemC-AMS extends SystemC for mixed signal modelling [5], adding support for signal flow models and conservative-law models. Modelica [6] is similar in being declarative, but it is only a modelling language; the simulation engine is unspecified. None of the tools researched (in [2], [5]–[9]) provide a higher abstraction in the form of modelling signals as functions of time and separating the notions of time. Ptolemy and SystemC-AMS do 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 agree. SystemC-AMS supports module and port time step propagation, but as a consistency check. Ptolemy also support higher order components, but not higher order signals.

The closest to our approach is Functional Reactive Programming (FRP) [8], [9]. FRP also uses functions of time (originally behaviours) implemented in Haskell, but for the CT and reactive domains, and has made excellent progress in applying it for different applications and providing formal semantics. However, in contrast to our approach, FRP does not identify and use different notions of time and in the reactive domain data values are aligned with a global clock, while we allow arbitrary sample rates in the DT domain.

VII. CONCLUSION

We have presented a novel approach for modelling mixed CT and DT systems with signal flow diagrams. By representing signals as function of time, we can exactly determine the value of a signal while retaining efficiency, even with multiple time delays or for a multi-rate system.

The second contribution is that we separated the different notions of time, thereby allowing local time adjustments. Further, the time step granularity is only propagated to its inputs and not the rest of the system, e.g. each integral can use a different time step for approximation. Thirdly, the inputs of ADCs are only evaluated at the sample time and these times are independent.

Interesting future work would be to implement symbolic integration with a fall-back to numeric integration, hidden in the composition operators.

REFERENCES

- [1] E. A. Lee, "Cyber physical systems: Design challenges," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-8, Jan 2008. [Online]. Available: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html>
- [2] L. P. Carloni, R. Passerone, A. Pinto, and A. L. Angiovanni-Vincentelli, "Languages and Tools for Hybrid Systems Design," *Found. Trends Electron. Des. Autom.*, vol. 1, no. 1/2, pp. 1–193, 2006.
- [3] K. C. H. Blom, M. D. van de Burgwal, K. C. Rovers, A. B. J. Kokkeler, and G. J. M. Smit, "DVB-S signal tracking techniques for mobile phased arrays," in *Vehicular Technology Conference Fall (VTC 2010-Fall), 2010 IEEE 72nd*, Ottawa, ON, Canada. USA: IEEE Vehicular Technology Society, September 2010, p. 5.
- [4] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity - the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, Jan 2003.
- [5] A. Vachoux, C. Grimm, and K. Einwich, "SystemC-AMS Requirements, Design Objectives and Rationale," in *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2003, p. 10388.
- [6] P. Fritzson and V. Engelson, "Modelica - A Unified Object-Oriented Language for System Modeling and Simulation," in *ECOOP'98 — Object-Oriented Programming*, ser. Lecture Notes in Computer Science, E. Jul, Ed. Springer Berlin / Heidelberg, 1998, vol. 1445, pp. 67–. [Online]. Available: <http://dx.doi.org/>
- [7] H. Zheng and E. Lee, "Operational Semantics of Hybrid Systems," *EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2007-68*, May, vol. 18, pp. 2007–68, 2007.
- [8] C. Elliott and P. Hudak, "Functional Reactive Animation," in *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*. New York, USA: ACM, 1997, pp. 263–273.
- [9] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, "Arrows, Robots, and Functional Reactive Programming," in *Summer School on Advanced Functional Programming 2002*, Oxford University, ser. Lecture Notes in Computer Science, vol. 2638. Springer-Verlag, 2003, pp. 159–187.