Multi-domain transformational design flow for embedded systems

Kenneth C. Rovers, Marcel D. van de Burgwal, Jan Kuper, 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, M.D.vandeBurgwal, J.Kuper, A.B.J.Kokkeler, G.J.M.Smit}@utwente.nl

***Abstract*—Current tools for embedded system design have limited support for modelling the interaction of the system with its physical environment. Furthermore, the natural representation of (streaming, real-time) applications with dataflow models is not supported by most tools. However, integrating multiple domains supports the design of complex interdisciplinary systems and enables model transformations.**

In this paper we discuss a unified approach, called *UniTi*, to handle continuous and discrete time models in a single framework, which includes the dataflow model as well. Our approach consists of a transformational design flow, expressed mathematically in a functional language. We formally distinguish the various domains and explain their interaction. In addition, we give guidelines for specifying algorithms such that these transformations can be applied. Our approach is illustrated with a non-trivial case study: beamforming in a phased array system.

***Index Terms*—embedded system, system design flow, model-based design, model transformations, multi-domain**

I. Introduction

Designing, modelling and verifying embedded systems is a big challenge; one of the key problems being the interaction of the system with the physical world (its environment) leading to different views on for example time. A second problem consists of the strong requirements concerning the correctness and robustness (including dependability) of the software and architecture. Consequently, the need to design embedded systems in an integrated approach, including that correctness is verifiable, is widely recognised [1]–[3]. Furthermore, when designing such complex systems it is useful to apply model-based design, i.e. the iterative and incremental development of a single reference model, because it shortens the design cycles and integration is part of the design process early on.

In this paper we present an approach which supports the design process on these aspects, filling in a gap that is left by current design tools (such as [3]–[5]) which are able to solve this challenge only partially. The approach we choose 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 continuous time (CT), discrete time (DT), and dataflow modelling (DF). In the context of this paper, in which we limit ourselves to streaming applications running on tiled multi-core architectures [6], the CT and DT domains are mainly relevant for the hardware side of an embedded system, and the DF domain deals with the software side. Since functional languages are close to mathematics, we express the framework and models in the functional language Haskell.

As an example to illustrate the necessity to integrate the various domains we mention an ADSL modem, which uses adaptive transmission based on cable conditions. Current design tools implement the interaction between the above mentioned domains by discretising a global simulation time representing signals as a sequence of values, which prevents exact transformations with respect to time such as variable time delays (see [7]). Thus, based on these tools it is not possible to exactly verify the correct operation of the coding, modulation and error-correction without first developing a hardware prototype, because we can not differentiate between the error caused by the modeling tool and an error caused by an incorrect specification or implementation. Clearly, this is far too late. In order to solve this shortcoming, the CT domain, which deals with the conditions of the telephone cable, and the DT and DF domains, which deal with the computational aspects of the design, have to be analysed in an integrated way.

Currently there are no tools which model continuous time exactly. There are only few tools which are able to integrate the DF domain with the CT and DT domains, and there are even fewer tools which support model transformations in a multi-domain setting (more details can be found in section II). Because of the integrated approach presented in this paper we can apply model-based design using transformation steps, thereby guaranteeing the correctness of the design. We present a design flow in which the design steps correspond to a transformation of the model using the multi-domain framework. In particular, the contributions of this paper will be that we:

- identify and formally distinguish the domains (sec. III),
- provide the connections between the domains as part of a framework (sec. III),
- present an design flow to exploit this framework (sec. IV),
- present guidelines for transformations between the domains to enable model based design (sec. V),
- show flexibility in (partly automated) transformations to enable design space exploration (sec. VI) and
- illustrate the design flow with a case study (sec. VII).

II. Related work

There are many multi-domain simulation tools. For an extensive survey see [3]. The most well-known is MATLAB/Simulink, supporting mixed CT/DT modelling and Stateflow for finite state machines. Ptolemy [4] supports many more

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

domains, including CT and DF and experimental DT support, with the goal of researching their interaction. 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 multi-domain signal flow models, also with dataflow semantics. All these tools implement the CT domain by solving a set of differential equations using a global time step, 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 [7]. Our approach applies exact time transformations without the need for a solver.

Furthermore, support for model-based design from current tools is lacking. SystemC-AMS has no support for modelling transformations. Ptolemy only recently added initial support for model transformations using higher-order components [8], a similar but less flexible approach than described in this paper. Other projects researching automated system level modelling with model transformations are Sesame [9] and Daedalus [10]. Their focus is on automatically parallelising applications, with the restriction that they consist of static affine nested loop programs. Our approach is more flexible, and intuitive in specifying models and in applying higher-order model transformations, because of its mathematical basis.

Our multi-domain design flow is supported by a framework in a functional language. A key feature exploited is the use of higher-order functions, used for model interactions and model transformations. The field of Functional Reactive Programming (FRP) [11], [12] also uses functions of time (originally behaviours) for CT modelling and has made excellent progress in applying it to different domains and providing formal semantics. In [13] higher-order model transformations for parallelization are introduced as strategies to complement an algorithm. Our framework applies the best aspects from these approaches to multi-domain model-based design.

III. Formalisation of the domains

In this section we present a unified formalisation of the continuous time (CT), the discrete time (DT) and the dataflow (DF) domains such that components in these domains can all be specified in the same formalism. We call our approach *UniTi*, emphasising the unification is based on time.An advantage of our approach is that there is no need to discretise continuous signals. Further, the integration of DF in the same model is a major advantage.

We also define composition operators (for sequential, parallel, and feedback composition) which are valid for all three domains, leading to a flexible modelling of the system under design, thus supporting model transformations and design space exploration. These composition operators allow for a block diagram like specification of the design. Moreover, since we use a functional language, simulation of the design is done by straightforwardly evaluating the model. This saves us from the need to develop a specific solver (i.e. equation system and solution algorithm), as is standing practice in current tools.

A. Domains

We start with a note on terminology: in all three domains we use the term *signal* for a connection between components, representing varying data over time or space as standard in engineering. Correspondingly, a *component* is a transformation of (incoming) signals into (outgoing) signals. Thus, we start with the definition of data types for components and signals in the three domains.

CT: 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:

$$Time = \mathbb{R}$$
$$Sig_{CT} = Time \rightarrow \mathbb{R}$$
$$Component_{CT} = Sig_{CT} \rightarrow Sig_{CT}$$

Note that $Component_{CT}$ is a “higher order type”, i.e. it has a function (of type Sig_{CT}) as argument and delivers in another function (also of type Sig_{CT}) as result, thus expressing that a component transforms signals. In combination with the fact that the continuity of time is immediately present in the type definitions as well, this makes it possible to express the fact that a component can locally control or change the time reference of a signal. For example, consider a component $delay_\delta$ which applies an arbitrary time delay δ to a signal f:

$$delay_\delta(f) = t \mapsto f(t - \delta) \tag{1}$$

(where the notation $x \mapsto ..x..$ denotes the function which maps x to $..x..$). Note that $delay_\delta$ indeed is of type $Component_{CT}$.

DT: The DT domain is concerned with the digital hardware (such as a FIR filter) of a system in which a *signal* is represented by (a sequence of) samples, linked to discrete sample times. A component produces output values dependent on a single input sample and possibly on previous inputs. In order to express the influence of the history of the processing, a component has 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. Hence, a component can still be seen as a signal transforming function.

This leads to the following type definitions:

$$Sig_{DT} = \mathbb{R}$$
$$Component_{DT} = Sig_{DT} \rightarrow Sig_{DT}$$

It is important to note from the perspective of a component a signal now is *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, thus from the perspective of the component time is abstracted away from. Note that the type of components has the same structure as in the CT domain.

An example of a DT component is adding a constant n to a signal x (as in a bias, or a level shifter):

$$add_n(x) = x + n \tag{2}$$

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

DF: A dataflow model is a graph of nodes (processes) connected by edges (channels); data tokens are processed inside nodes and sent from one node to another through the edges. Tokens are abstract in the sense that they may have any internal structure. A process may consume and produce several tokens at a time; when there are not enough tokens available on the input edges of a node, that node will not execute (fire).

Dataflow modelling is used for representing software for multi-core systems, especially when real-time guarantees are required. It offers methods to analyse deadlock and race conditions, to calculate buffer sizes, and to determine or estimate latency and throughput of tokens streaming through the graph (see [14]). Further, the graph structure of a dataflow model allows for mapping an application to a multi-core system.

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 then is 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. Note that this is slightly different from the standard implementation of dataflow in which channels store tokens and connect processes, while we use signals for connections and store (input) tokens in a component with the process.

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 $Tokens$):

$$Sig_{DF} = [Token]$$
$$Component_{DF} = Sig_{DF} \rightarrow Sig_{DF}$$

Here too, the structure of a component is the same as before. Note that according to this definition a component transforms a signal each time it receives a signal, also in case it did not collect enough tokens to fire. To model that, it is possible a component sends an empty signal containing zero tokens. The other way around, i.e. a signal contains more tokens than a component needs in order to execute, 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 a design only the functionality of the processes and the connection between them have to be specified. Our framework takes care of managing channels, firing rules and execution.

As a simple example, we consider a DF component that calculates the average of three numbers (tokens), i.e. the functionality of the DF process is expressed in the definition of the function $mean_3$:

$$mean_3\ (x, y, z) = (x + y + z)/3$$

Clearly, this process can only execute when there are three values available, and dependent on the number of tokens in the incoming signal, the contents of the internal state may change. Suppose, initially there are two tokens (2,4) in the state. The $average$ component as a whole, i.e. its functionality together with its initial state, is now formulated as[1]:

$$average = mean_3 \Uparrow [2, 4] \tag{3}$$

Our framework takes care that when the component $average$ is applied to a signal s, it will add the tokens from s to its internal state, then apply the function $mean_3$ as many times as possible (possibly zero times), each time removing three tokens from the state, and finally packing the results in a signal to be sent. Thus, after firing the state of $average$ is automatically updated by our framework.

B. Composition

All the domains above have components that take an input signal to produce an output signal, so we can provide generic rules for composition in these domains. *Sequential* composition (figure 1) is defined as:

$$\varphi \triangleright \psi = f \mapsto \psi(\varphi(f)) \tag{4}$$

with $\triangleright$ the operation that takes the output signal of component φ as input signal of ψ and returns a new component with the input signal (f) of φ and the output signal of ψ.

Likewise, parallel composition (figure 2) is defined as:

$$\varphi \parallel \psi = (f, g) \mapsto (\varphi(f), \psi(g)) \tag{5}$$

i.e. multiple inputs are represented as tuples and the composition connects φ to the first and ψ to the second.

Feedback composition (figure 3) is defined as:

$$\circlearrowleft \varphi = f \mapsto g, \ \text{ where } \ (g, h) = \varphi(f, h) \tag{6}$$

i.e. the component φ takes two inputs f and h, of which h is the second output signal of itself, i.e. a signal that is fed back.

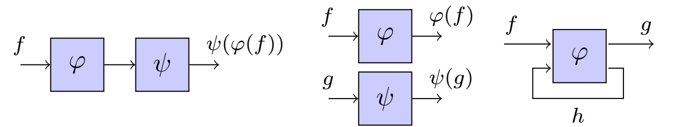

Fig. 1. Sequential

Fig. 2. Parallel

Fig. 3. Feedback

C. Multi-domain integration

More importantly, our framework offers composition of mixed CT, DT and DF components for a multi-domain simulation. Technically, this is achieved by the framework 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. Because we have local control over time in a CT component, this gives the advantage that simulation can be done without loosing efficiency [7].

[1]The $\Uparrow$ operator is defined using the same abstraction as the state in the DT domain, in combination with a continuation. It falls outside the scope of this paper to discuss the details of this mechanism.

DT ⇒ CT: To embed a DT component into a CT component it must accept a function of time instead of a single value (see section III-A) . This is straightforward, since the DT operation is independent of time. For example, take the DT addition:

$$add_n(f) = t \mapsto f(t) + n$$

where instead of a single input value x, an input function f is used, yielding the function for which n is added to the result for every t. This conversion is automatic when necessary.

The boundary between the CT and DT domain is the analogue-to-digital converter (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 (with d the sample period):

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

Our framework automatically adapts the DT to the CT domain, such that the composition operator $\triangleright$ is able to connect components from these domains to each other.

DF ⇒ DT: To change a DF component to a DT component, it must accept single values instead of a list of tokens. Without going into details, we remark that 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. Because the DT values are linked to a sample time, the DF process now is extended with time while on its own it only models ordering of tokens. Therefore, execution time of a DF process also has meaning; the produced tokens after execution are considered values in the DT domain (with a delayed sample time because of the execution time).

Here too, the $\triangleright$ operator is able to connect components from the DT and DF domain to each other by packing or unpacking signals. Now components from all domains can be composed by taking the DT domain as an intermediate step.

D. Implementation

The implementation of components is straightforward from the mathematics, because we use the functional language Haskell. For example, the $delay_\delta$ (eq. 1) and add_n (eq. 2) components are implemented as:

```
delay delta f  =  \t -> f (t-delta)
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`) follows later when the system is evaluated. This is called *partial application*.

The DF component $average$ (eq. 3) is implemented as:

```
average = mean_3 ^^^ [4,2]
```

where the `^^^` operator is the Haskell formulation of the $\Uparrow$ operator used before.

The composition operators of eq. 4, 5 and 6 follow as:

```
phi >>> psi = \f      -> psi (phi f)
phi ||  psi = \(f, g) -> (phi f, psi g)
loop phi    = \f      -> let (g, h) = phi (f, h) in g
```

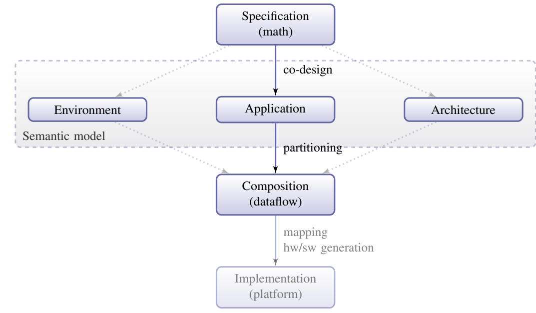

Fig. 4. Design flow for tiled architectures

These definitions are exactly the same as in mathematics, except that the arguments `f`, `g` and `h` are not written between brackets. Note that we require lazy evaluation to allow `phi` to break the recursive dependence on `h` in `loop`. The operators are overloaded in case different domains are composed. This implementation is more involved and is left out for brevity.

An example of a mixed domain system then follows as:

```
system = delay 0.1 >>> adc 0.3 >>> add 1 >>> average
```

where `delay 0.1` and `adc 0.3` are CT components, `add 1` is a DT component, `average` is a DF components, and their composition `system` is a CT component.

E. Final remarks

Clearly, the above only presents single input, single output components whereas realistic systems may contain multiple input, multiple output components. By a straightforward generalisation of the above description, our framework is able to deal with multiple input, multiple output components as well. However, for brevity reasons we do not discuss that here.

Integration and differentiation are implemented using one of the standard numerical approximations. This approximation is implemented as a signal transformation and as such a "normal" part of the model when evaluated. Therefore, a dedicated solver is not needed and the developer can choose the complexity and accuracy of the approximation per instance.

IV. Design flow

Using the domains defined in the previous section, we will present a transformational design flow to identify and guide typical steps encountered when designing embedded systems. Iterative, verifiable steps transform a single model into a division of functionality over the environment, the architecture (analogue and digital hardware) and the application (software), as well as a partitioning of the software over multiple cores. Although these steps described may be well-known, it is important to match them with the presented domains and with model transformations. A connection that is not trivial, as evident from the lack of support in current tools.

Figure 4 illustrates the flow. The rounded rectangles represent models and the arrows represent transformations. The multi-domain model including the environment, the architecture and the application is called the *semantic model*.

The design flow uses a top-down divide-and-conquer approach. The initial (functional) specification of a system is

readily implemented and verified in the CT domain. We will discuss the co-design and partitioning steps; the mapping and hardware/software generation are beyond the scope of the present paper. Co-design can be seen as a division *over* the domains, while partitioning can be seen as a division *within* a domain (the DF domain in particular).

A. Co-design

During the co-design process, functionality is divided over the different domains. We distinguish a number of tasks:

- Decide what is needed for simulation and verification from the environment. The environment is modelled in the CT domain.
- Define the architecture and decide what is implemented in analogue hardware (CT domain) and what in digital hardware (DT domain).
- Decide what to do in fixed hardware (ASIC) and what to do in programmable hardware and software (DF domain), thereby refining the architecture and defining the application.

B. Partitioning

After functionality is assigned to hardware or software, the software is partitioned over the programmable hardware (cores) in case of a multi-core architecture. The performance and efficiency of the software is determined by computation and communication costs. The computation is the actual work to be done, while the communication ensures the data is available at the right place. Having the data close to the computation, increases the efficiency by lowering the communication costs, exploiting so called "locality of reference". As partitioning separates the computation, it introduces extra communication and has an influence on the performance.

We use a dataflow model where the processes contain the functionality and the communication is made explicit via channels. Such a model thus represents a partitioning of the software and transforming the dataflow graph changes the partitioning. Execution and channel content can be monitored with our framework and the graph can be analysed, although analysis is currently not implemented.

V. Model transformations

The previous section presented a design flow for dividing functionality over the domains. Our multi-domain framework allows for a single model during the design process. However, a particular difficult aspect of model-based design that we have not discussed are model transformations. We will discuss model transformations for the co-design and partitioning steps.

A. Co-design

Much of the co-design process is automated. A mixed CT and DT model is transformed from a CT model by only adding an ADC. 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 is the same. Furthermore, the conversion between domains is also automated by the composition operator, except for the sample period which is explicitly added with an ADC (see section III).

For example, a domain independent definition of an addition of 1 (bias) followed by a multiplication with 0.12 (gain) is:

$$(+1) \ \triangleright \ (*0.12)$$

where the input signal determines wether functions of time, values, or tokens are added and multiplied. However, by only adding an ADC, the addition is in the CT domain while the multiplication is in the DT domain:

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

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.

B. Partitioning

Automated software partitioning into parallel processes is more involved. We identify two kinds of parallelism: data parallelism and control parallelism. In the first the data is split. Examples are bit-level and data-level parallelism. In the second the control, i.e. the operations on the data, is split. Examples are instruction, task and pipeline parallelism.

Control parallelism: Control parallelism occurs when some operations 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. Thus, the functionality, the inputs and the outputs are made explicit to fit to the dataflow model and are wrapped in a DF component.

In the DF domain, data in channels must remain ordered. A process can only execute when there is enough space in its output buffers[2]. Thus, a process is stalled if the tokens are not consumed from the output buffer fast enough by the next process. This is called "back-pressure" and results in automatic synchronisation in parallel execution.

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.

[2]Channels are of unbounded capacity, but buffers between processes are modelled by two channels in opposite directions; one carries the tokens to be communicated and the back-edge models empty space in the buffer.

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 prefer 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:

$$\vec{a} \bullet \vec{b} = \sum_{i=0}^{n-1} a_i \cdot b_i \tag{7}$$

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 step by step the results 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 [9], [10].

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. 7, it can be seen that we need:

- pairwise multiplication of the elements of the vectors. We use the notation $\hat{\cdot}$ for this lifted version of multiplication,
- the reduction of the resulting values to a single value by using $+$. We use $\bigoplus$ to denote this interpretation of addition, i.e. the expression $\bigoplus \vec{x}$ 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:

$$\vec{a} \bullet \vec{b} = \bigoplus (\vec{a} \,\hat{\cdot}\, \vec{b}) \tag{8}$$

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 $\hat{\cdot}$ and $\bigoplus$ are now visible on aggregate level. In the algorithm for the dot product these operations are *separated*.

The $\hat{\cdot}$ operates on the data independently, so it is easy to parallelise. However the reduction operation $\bigoplus$ must be associative to be able to use parallelism. For example, splitting vector $\vec{a}$ and $\vec{b}$ in three sub-vectors $\vec{a}_0, \vec{a}_1, \vec{a}_2$, respectively $\vec{b}_0, \vec{b}_1, \vec{b}_2$ (where $\vec{a}$ and $\vec{b}$ are equally long) leads to the following parallelisation of the dot product:

$$\begin{aligned} e_i &= \bigoplus (\vec{a}_i \,\hat{\cdot}\, \vec{b}_i), \quad \text{where } \ i = 1, 2, 3 \\ \vec{a} \bullet \vec{b} &= \bigoplus [e_0, e_1, e_2] \end{aligned} \tag{9}$$

Note that the calculation of e_i and $\bullet$ 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 [13].

Automation: 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. 8 to eq. 9 is automated (the reduction operation must be associative). For that example, four processes are generated, one each for e_1, e_2 and e_3 and one for $\bullet$. More parallel strategies can be found in [13].

The granularity (the vector is split in three in our example above) is a manually specified parameter. The amount computation and communication per process must be matched with the capabilities of the processors and the network.

VI. Design space exploration

We have touched upon a number of trade-offs that the designer must specify:

- analogue/digital co-design
- hardware/software co-design
- the way the functionality is specified
- the granularity of the partitioning

As these are trade-offs, it is very helpful for a designer to try a few different alternatives, so called design space exploration. With the help of the multi-domain framework, in many cases it is simply a matter of moving the ADC or applying a different parallelisation strategy.

By using aggregate operations the model is independent from the number of data elements. However, the number of data elements does influence the number of processes or the amount of computation and communication per process. As the partitioning is automated we can quickly explore the results with different number of elements and different granularities.

VII. 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.

A. 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 figure 5. 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.

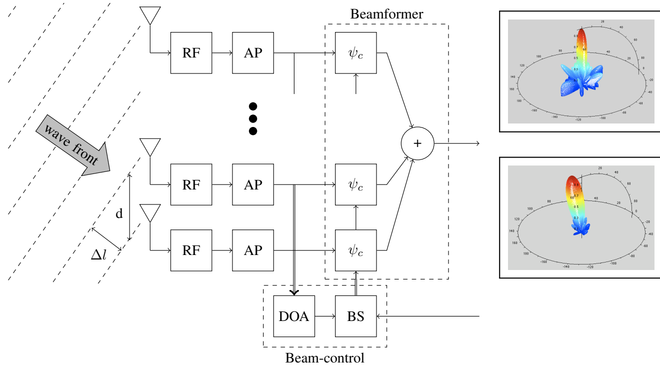

Fig. 5. Phased array receiver

Beamforming is based on interference. A wavefront arrives at different times at the antennas because of the path length difference (see figure 5). 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 implemented as:

```
systm f = sum [(delay p d f) * (correct p b f) | p <- ps]
```

where `p` is an element position (`Pos`), `d` is a direction of arrival (`DOA`), `b` is a beam-steer direction (`BSt`) and `f` is the source signal. The implementation uses complex numbers to calculate the `delay` of the source and apply the `correct`ion for each antenna element position from `ps`.

B. Co-design (Semantic model)

To validate the phased array receiver, the signals received at the antennas are generated, representing the environment. Therefore, a source, a transmitter and a channel are added. The model of the channel implements the delay from the source to the different receiver antennas.

The intended architecture is a tiled multi-core SoC for the beam-control and beamformer 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. During the co-design step, beamforming is represented as a complex multiplication and addition in the DT domain. For brevity the beam-control processing is left out. See [15] for more about beam-control processing on a multi-core platform.

A block diagram of the resulting simulation model is shown in figure 6. Also the function names and the hierarchical composition, discussed next, are indicated in the figure.

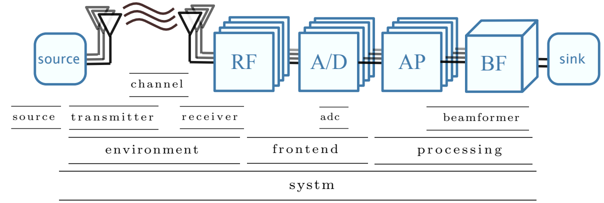

Fig. 6. CT/DT co-designed system block diagram

a) Signal flow model implementation: The top level of the division from the co-design process is represented as:

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

Here, the channel `delay` from the specification is part of the `environment` and the `correct`ion is part of the processing. These function are defined to use aggregate data structures, i.e. the input of the `environment` is a list of sources and the output a list of antenna signals:

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

A separate channel with a different delay is applied for each source and antenna element combination. 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.

The `transmitter`, `channel` and `receiver` functions are defined at the element level. 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 -> SigCT -> SigCT
channel p f = \d t -> f d (t+dt)
   where
      dt = (delay p d)/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))
```

The `frontend` consists of an `rf` down-conversion and an `adc` for each receiver:

```
frontend :: [SigCT] -> [SigDT]
frontend = mimo rf >>> mimo (adc d)
```

The processing consists of a FIR filter (`ap`) and a `beamformer` per beam:

```
processing :: [Pos] -> [BSt] -> [SigDT] -> [SigDT]
processing ps bs = mimo ap >>> simo (beamformer ps) bs
```

The `beamformer` applies the dot-product (`**`) of the input signals with the correction vector:

```
beamformer :: [Pos] -> BSt -> [SigDT] -> [SigDT]
beamformer ps b ss = cs ** ss
   where cs = [correct p b | p <- ps]
```

The dot-product is thus implemented at the aggregate level as explained in section V.

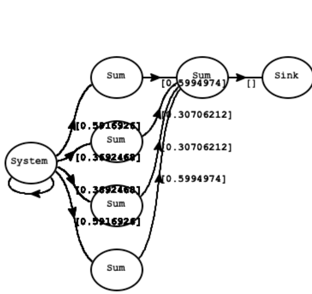

Fig. 7. 16 antenna beamformer

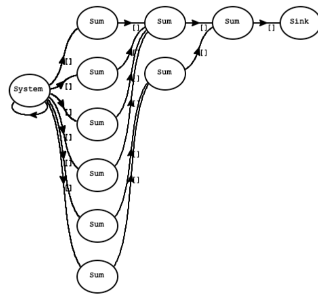

Fig. 8. 23 antenna beamformer

b) Radiation pattern: As an example of the flexibility of the aggregate definitions, we reuse the components from the signal flow implementation to generate radiation patterns. This is achieved by calculating the transfer function over all angles.

Figure 5 shows a radiation pattern with a 3 by 5 element array, no beamsteering and arbitrary placed elements in the top-right. This results in a flattened beam at 0° inclination and somewhat chaotic side lobes. The bottom-right shows a 4 by 4 regular array with a (20°,110°) steering angle.

C. Partitioning (Dataflow model)

The beamformer combines all the antenna signals and is a computationally intensive block of the system. 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 section V is used to partition the beamformer. Furthermore, the beamformer specification is independent from the number of antennas.

The intended core is a 200MHz 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 50MSPS. 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 figure 7) and thus 5 cores. For 23 antennas the beamformer consists of 9 processes, shown in figure 8, which matches a 3x3 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.

VIII. Results

In this section we will evaluate the applicability and flexibility of the design flow and the supporting framework.

A. Applicability

Figure 9 shows the complete framework during the simulation of the beamforming case study with a 5x5 array, a 30° steering direction and two sources, one of which is filtered away. It shows the CT, DT and DF domains in a single model, including simulation and multiple hierarchical levels.

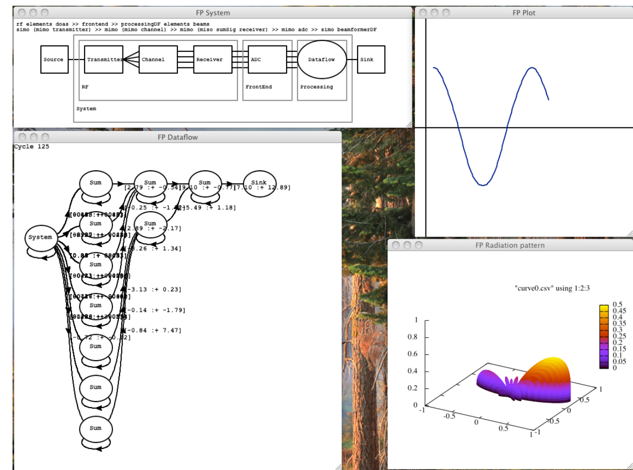

Fig. 9. Framework

1) Performance: Profiling results for an increasing number of antennas (3x3, 5x5 and 11x11 arrays) are shown in figure 10. The top two show the result of the semantic model. The next two include the radiation pattern. The final two also include the dataflow model. We can see that the memory requirements grow faster with more antennas then the processing requirements. This limits the simulation to a few hundred antennas for a 2GHz Core2 Duo with 4GB RAM. Further, the radiation pattern calculation becomes dominating with larger arrays. The instantiation of the wxWidgets toolkit for the GUI has a relatively large but fixed processing overhead compared to the model. With the dataflow model the GUI becomes dominating and is also dependent on the array size, because of the redrawing of the Bezier curves and the channel contents use a naive implementation. This is the first candidate for optimisation. Comparison of the case study (without the dataflow model) with an implementation in Simulink gives ±10 times speed-up in our advantage [7]. 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.

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 I shows the code size in lines. We see that the framework is 2500 lines, half of which is of the dataflow simulator. The case is about 600 lines, with the majority in the semantic 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 overhead, in code of the case study needed because of the design framework, is very little, about 5%.

It is difficult to estimate and compare the development time of the case study, as it was used to develop to the framework. 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

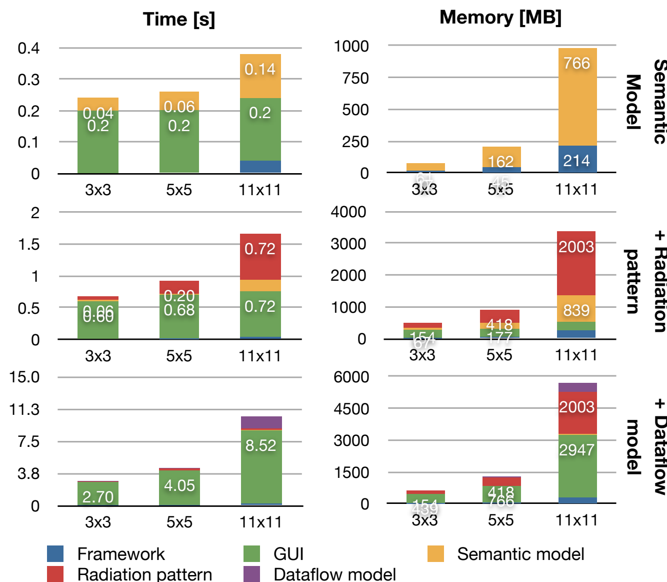

Fig. 10. Profiling results (time [s] and memory [MB]) for the beamformer case study on a 2GHz Core2 Duo with 4GB RAM.

TABLE I
CODE SIZE [LINES]

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

improves flexibility (see below) making adjustment easier. Furthermore, changes are checked by the type system and transformations are defined to be correctness preserving.

B. Flexibility

The presented design flow and guidelines for specifying the algorithm 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.

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

2) 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 figure 10.

IX. CONCLUSIONS

We have presented a design flow for multi-core embedded systems supported by a multi-domain modelling framework and model transformations. This flow allows the designer to develop a mathematical specification to a division over the continuous time, discrete time and dataflow domains during the co-design step and a division of the software over processes in the dataflow domain during the partitioning step. We've provided guidelines on how to specify the functionality so that it becomes independent of the number of elements in the actual data types, thus we can apply model transformations for automated partioning. Furthermore, the framework covers all multi-domain integration aspects and the designer (almost) only has to specify the functionality of the components.

Evaluation of the phased array beamforming design using the presented design flow shows that the performance is good for reasonably large designs, being limited by the memory requirements during simulation. The framework hardly requires additional code from the designer, the overhead of code in the beamforming case caused by the framework is just about five percent. The resulting phased array design is flexible with respect to the number of antennas and their positions, the number of sources and the number of cores to partition over.

REFERENCES

[1] E. A. Lee, "Cyber physical systems: Design challenges," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-8.

[2] B. S. Blanchard and W. J. Fabrycky, *Systems Engineering and Analysis*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall, 1998.

[3] L. P. Carloni, R. Passerone, A. Pinto, and A. L. Angiovanni-Vincentelli, "Languages and Tools for Hybrid Systems Design," *Foundations and Trends in Electronic Design Automation*, vol. 1, no. 1, pp. 1–193, 2006.

[4] J. Eker *et al.*, "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 *Proceedings of the conference on Design, Automation and Test in Europe (DATE '03)*. IEEE, 2003.

[6] G. J. M. Smit, A. B. J. Kokkeler, P. T. Wolkotte, and M. D. van de Burgwal, "Multi-core Architectures and Streaming Applications," in *10th Int.W. on System-Level Interconnect Prediction*, Apr. 2008, pp. 35–42.

[7] K. C. Rovers, J. Kuper, and G. J. M. Smit, "The problem with time in mixed continuous/discrete time modelling," *ACM SIGBED Review*, vol. 8, no. 2, Jun. 2011.

[8] T. H. Feng and E. A. Lee, "Scalable Models Using Model Transformation," EECS Dep., U. of California, Berkeley, Tech. Rep. UCB/EECS-2008-85.

[9] C. Erbas, A. D. Pimentel, M. Thompson, and S. Polstra, "A framework for system-level modeling and simulation of embedded systems architectures," *EURASIP Journal on Embedded Systems*, vol. 2007, no. 1, Jan. 2007.

[10] H. Nikolov *et al.*, "Daedalus: toward composable multimedia MP-SoC design," in *DAC '08: Proceedings of the 45th annual Design Automation Conference*. ACM, 2008, pp. 574–579.

[11] 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.

[12] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, "Arrows, robots, and functional reactive programming," in *Advanced Functional Programming*, ser. Lecture Notes in C.S. Springer, 2003, vol. 2638, pp. 159–187.

[13] P. Trinder, K. Hammond, H. Loidl, and S. P. Jones, "Algorithm + Strategy = Parallelism," *J. of Functional Programming*, vol. 8, pp. 23–60, 1998.

[14] M. H. Wiggers, "Aperiodic Multiprocessor Scheduling for Real-Time Stream Processing Applications," Ph.D. dissertation, University of Twente, the Netherlands, Jun. 2009.

[15] M. D. van de Burgwal, K. C. Rovers, K. C. H. Blom, A. B. J. Kokkeler, and G. J. M. Smit, "Adaptive Beamforming using the Reconfigurable Montium TP," in *Proceedings of the 13th Euromicro Conference on Digital System Design*, Sep. 2010, pp. 301–308.