Composite Signal Flow: A Computational Model Combining Events, Sampled Streams, and Vectors

Axel Jantsch, Royal Institute of Technology, Sweden
Per Bjurèus, CelsiusTech Electronics AB, Sweden

Abstract

The composite signal flow model of computation targets systems with significant control and data processing parts. It builds on the data flow and synchronous data flow models and extends them to include three signal types: non-periodic signals, sampled signals, and vectorized sampled signals. Vectorized sampled signals are used to represent vectors and computations on vectors. Several conversion processes are introduced to facilitate synchronization and communication with these signals. We discuss the severe implications, that these processes have on the causal behaviour of the system.

We illustrate the model and its usefulness with three applications. A co-modelling and co-simulation environment combining Matlab and SDL; a high level timing analysis as a consequence of the operations on vectors; conditions for a parallel, distributed simulation.

1. Introduction

Current approaches to system modelling can be divided into two groups, homogeneous and heterogeneous models. Homogeneous models are based on a single formalism or language such as VHDL [8, 10], C [12], C++ [20], SpecChart [23], Java [14, 24, 27], SDL [7, 26], etc. These languages are rich and can typically be used far beyond their original scope, in particular when they are extended with special features, e.g. for communication. However, such homogeneous solutions come at a price. A language, which is well established in one community is not always well received in another community, both for practical and technical reasons. The modelling concepts of general purpose languages are not pertinent for the concepts of a given application problem. For the two domains of control-dominated systems and signal processing, it is difficult to find a language that naturally accommodates both worlds.

For these reasons heterogeneous frameworks have been proposed. They build on existing models and languages and devise techniques to integrate them. A very general and most influential framework is Ptolemy [4]. Ptolemy defines several models of computation, such as discrete event or data flow domains. It provides a general mechanism for communication between different domains. A mechanism for communication and synchronization between data flow and discrete event models has been implemented in Ptolemy, which transforms each single event on the border between the two domains. Indeed, most general frameworks, which provide heterogeneous modelling environments, are based on this principle of transferring single events between different domains, e.g. [3, 5, 19]. This is true for both backplane environments [4, 21] and for unifying internal representations [1, 15]. Furthermore, unifying internal representations are typically complex, not based on a formal semantics, and therefore not easily amenable to formal analysis.

The composite signal flow model is an attempt to provide a relatively simple, formally sound model of computation, which can be the basis for heterogeneous modelling and simulation environments. By providing three different signal types, i.e. non-periodic signals, sampled signals, and vectorized sampled signals, it accommodates both control and data flow dominated parts. It also provides for conversion processes between the different signal types, thus allowing for adequate communication and synchronization between different system parts. It pays particular attention to the problem of causality which occurs due to conversion between sampled signals and vectorized signals. Essentially, the problem arises because data flow and control flow parts have to be synchronized. On one hand, data processing is often efficiently and conveniently modelled and implemented in terms of operations on vectors rather than scalars. On the other hand, control events must occasionally be synchronized with a time instance “inside” the vectors, because the vectors correspond to a time period. This dilemma can be solved by chopping up the vectors into individual events corresponding to time instances, whenever synchronization between data and control parts is necessary. However, this sacrifices the efficiency and elegance of powerful vector operations available in languages like Matlab [22]. It also sacrifices correspondence to the implementation in those

cases, where vectors and their manipulations are an abstract but correct model of the implementation.

Our proposal is very similar in nature and motivation to the work done by Poigné et al. [25]. Similarly to us, they provide a sound semantic basis to combine control and data flow aspects in a system. They also allow for heterogeneous modelling in different languages. However, their work differs in two ways. First, Poigné et al. address purely synchronous models and provide a common semantic frame for the synchronous languages Esterel, Argos and Lustre. In contrast, we address asynchronous systems and our time model is essentially a continuous time with totally ordered events. Second, they do not deal with vectors of sampled streams, which is one of our main issues.

Much of the semantics of the processes in the composite signal flow model is derived from data flow and synchronous data flow process networks discussed by Lee et al. [16, 17]. We define signals according to the tagged-signal model proposed by Lee and Sangiovanni-Vincentelli [18]. Based on this, we propose a model which handles operations on sampled signals and on vectors. In particular, the transformation between the different signal types receives attention. In this way, we address the modelling of systems with both, significant control and data processing.

2. Formal Model

2.1 Signals, Sampled Signals, Vectorized Signals

Using the framework for timed models of computation introduced by Lee et al. [18], we define signals and processes of the composite signal flow model. Given a set of *values V* and a set of *tags T*, where $T = \Re$, the reals, we define an *event e* to be a member of $T \times V$. The tags are used to model time. A *signal s* is a set of events, hence $s \subseteq T \times V$. A functional signal is a possibly partial function from T to V. "Partial" means, that a function may only be defined for a subset of T. The term function has the usual meaning, i.e. a signal has at most one value for a given tag; if $e_1 = (t, v_1) \in s$ and $e_2 = (t, v_2) \in s$, then $v_1 = v_2$.

A *sampled signal* s^λ is a signal which has only values for tags, which are $\lambda \in \Re$ apart. If $e_1 = (t_1, v_1) \in s^\lambda$, then $e_2 = (t_2, v_2) \in s^\lambda \Leftrightarrow (t_1 = t_2 + n\lambda) \vee (t_2 = t_1 + n\lambda)$, with *n* a natural number. In contrast, a *non-periodic signal* can have values at arbitrary time instances.

A *vectorized signal* s_v has a *v* sized tuple of values for each tag, where it is defined. Be W_v the set of tuples of size *v* over the set of values *V*; we define a *vectorized event* e_v to be a member of $T \times W_v$. Then a vectorized signal s_v is a set of vectorized events, hence $s_v \subseteq T \times W_v$. Analogous to the definition of functional signals, a functional vectorized signal is a possibly partial function from *T* to W_v.

Note, that the concept of a sampled signal reflects the real situation of many data or signal processing systems, which receive and transmit data streams with data appearing at regular intervals. In contrast a vectorized signal is a modelling artefact, which is sometimes used for convenience or efficiency at higher levels of abstraction.

Note furthermore, that in this paper we only deal with sampled signals with a constant sample rate, and with vectorized signals with equal sized vectors. However, we do not constrain the occurrence of different rates and vector sizes on different signals. Thus, the entire discussion is also valid for arbitrary multi-rate systems.

2.2 Processes

We define processes analog to Lee et al. [18] as relations between sets of signals, with input signals constraining the behaviour of a process. Since the details are not important here, we refer the reader to [18] for the sake of brevity. Note however, that processes can be composed to compound processes, thus forming a hierarchy.

As execution model for processes we adopt the notion of the *data flow process* put forward by Lee and Parks [17] in the variant, which allows a process to have state. For processes which operate exclusively on sampled signals, we use the more specialised model of *synchronous data flow* [16], because it is significantly cheaper to implement. However, we deviate from data flow and synchronous data flow process semantics by defining a few specific, atomic processes, which are not causal. They are discussed in the following sections.

2.3 Vectorization

When modelling with vectorized signals, it is almost always mandatory to use them in combination with ordinary and sampled signals, in order to interface to the environment or to control parts. To this end, it is necessary to transform sampled signals into vectorized signals and vice versa. Since these transformations are not always causal, we have to investigate the conditions of causality and when we can afford to violate them.

In the following we consider two variations of transformations between sampled signals and vectorized, sampled signals.

Head vectorization Ψ_n^h is a process which transforms a sampled signal s^λ into a vectorized sampled signal $s_n^{n\lambda}$ such, that *n* consecutive values in the sampled signal are compressed into a vector of size *n*. The tag of the vector in $s_n^{n\lambda}$ is identical with the tag of the first of these values in the sampled signal s^λ. More formally, if $e = (t + \lambda i, v_i') \in s^\lambda$ and $e_n = (t, w_n) \in s_n^{n\lambda}$ with $w_n = \langle v_0, \ldots, v_{n-1} \rangle$, then $v_i' = v_i$ for all $0 \le i \le n - 1$.

Tail vectorization Ψ_n^t produces a vector synchronized with the last of its corresponding element values in the input signal. *Head* and *tail de-vectorization* are the inverse processes such, that $\Omega_n^h(\Psi_n^h(s^\lambda)) = s^\lambda$ and $\Omega_n^t(\Psi_n^t(s^\lambda)) = s^\lambda$ (figure 1).

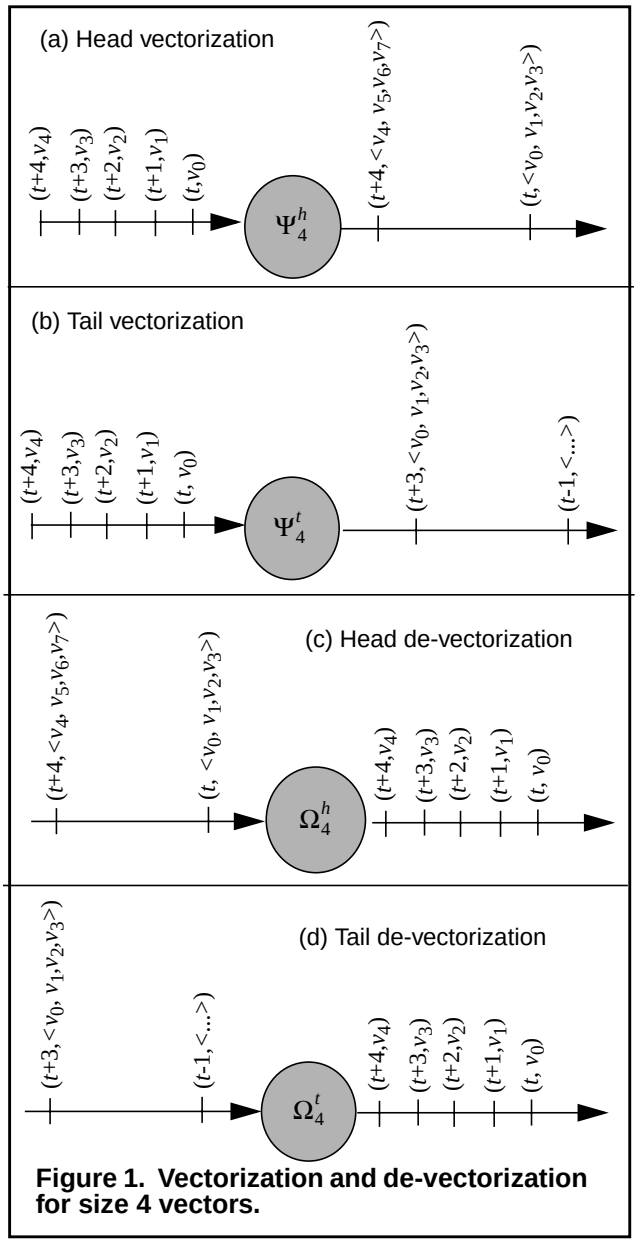

Figure 1. Vectorization and de-vectorization for size 4 vectors.

2.4 Conversion and Synchronization

For signal conversion and synchronization between sampled and non-periodic signals, i.e. between data flow and control parts, we define three processes: Pin_E, Pin_D and Punch.

A *pin encoding* process Pin_E transforms a non-periodic signal s into a sampled signal s^λ, $\mathsf{Pin}_\mathsf{E}(s) = s^\lambda$ such that,

$$e = (t, v) \in s^\lambda \Rightarrow \exists e' = (t', v) \in s \text{ with } (t' \le t) \text{ and } (e'' = (t'', v'') \in s \Rightarrow ((t'' \ne t) \text{ or } (v = v'))) \quad (1)$$

Intuitively, pin encoding takes all values of s and assigns them tags such, that they fit the sampling instances of s^λ. If there is more than one event in s between two sampling instances, only the last event is used. If there is no event between two sampling instances, the last value is repeated.

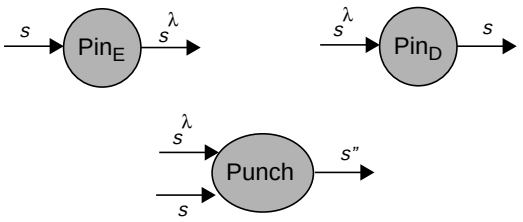

Figure 2. Pin-encode, Pin-decode and punch events.

A *pin decoding* process Pin_D transforms a sampled signal s^λ into a non-periodic signal s, $\mathsf{Pin}_\mathsf{D}(s^\lambda) = s$ such that,

$$e = (t, v) \in s \Rightarrow \exists e \in s^\lambda \text{ such that } e' = (t - \lambda, v') \in s^\lambda \Rightarrow v \ne v' \quad (2)$$

Intuitively, pin decoding takes all events from the sampled signal s^λ and places it into s but filters out repeated values.

A Punch process transforms a sampled signal s^λ into a non-periodic signal s', with events on the second input signal s defining the time tags in s'. $\mathsf{Punch}(s^\lambda, s) = s'$ such that,

$$e = (t, v) \in s' \Rightarrow \exists e' = (t, v') \in s \text{ and } \exists e'' = (t'', v) \in s^\lambda \text{ with } t'' \le t < t'' + \lambda \quad (3)$$

Intuitively, punching accesses the sampled signal s^λ at time instances defined by events in s. It is a synchronization mechanism between control and data flow parts.

2.5 Causality

In the following we need a delay process Δ_d, which delays every event on the input signal by a constant value $d \ge 0$, as illustrated in figure 3.

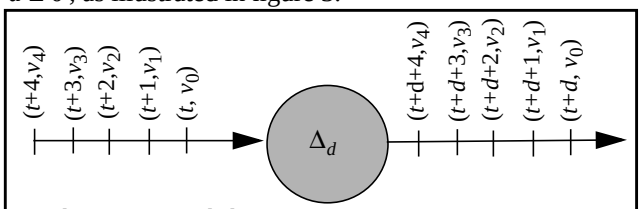

Figure 3. A delay process.

We follow [18] in the definition of causality based on the following metric. The distance between two signals s and s' is defined as

$$d(s, s') = \sup\left\{\frac{1}{2^t} : (s(t) \ne s'(t), t \in T)\right\} \quad (4)$$

The distance between two signals is the greater the earlier they differ from each other. The distance between two identical signals is 0, and for two signals differing already at the smallest defined tag it is infinite. This distance satisfies the conditions of a metric [6, 18]. Thus, the set of

signals together with this distance forms a metric space [6].

Now we can define that a process F is causal if, for any input two signals s and s', two output signals never differ earlier than two input signals, i.e.

$$d(F(s), F(s')) \le d(s, s') . \quad (5)$$

This formula can be generalized for processes with several inputs and outputs, but stating that the minimum distance of any of the input signals must be greater or equal to the maximum distance of any of the output signals.

We observe, that tail vectorization and head de-vectorization are causal, but head vectorization and tail de-vectorization are not causal.

$$\Psi_n^h \quad \text{... is not causal} \quad (6)$$

$$\Psi_n^t \quad \text{... is causal} \quad (7)$$

$$\Omega_n^h \quad \text{... is causal} \quad (8)$$

$$\Omega_n^t \quad \text{... is not causal} \quad (9)$$

We note further, that

$$\Omega_n^h(\Psi_n^h(s)) = s \quad \text{... is causal} \quad (10)$$

$$\Omega_n^t(\Psi_n^t(s)) = s \quad \text{... is causal} \quad (11)$$

$$\Psi_n^h(\Omega_n^h(s^\lambda)) = s^\lambda \quad \text{... is causal} \quad (12)$$

$$\Psi_n^t(\Omega_n^t(s^\lambda)) = s^\lambda \quad \text{... is causal} \quad (13)$$

$$\Delta_n(\Psi_n^h) \quad \text{... is causal} \quad (14)$$

$$\Delta_n(\Omega_n^t) \quad \text{... is causal.} \quad (15)$$

Hence, a combination of processes may be causal, even if not all of the constituting processes are causal. In fact, we can obtain a causal process from any non-causal process by combining it with an appropriate delay process. For a non-causal process A, the smallest n for which $\Delta_n(A)$ is causal, is called the *severity level* of A; such a process is denoted as $A_{s=n}$. This is a convenient way to express the conditions for models with different sampled and vectorized signals, perhaps with different sampling rates and vector sizes. The following processes are all causal under the given conditions (gcd denotes the greatest common divisor).

$$\Delta_d(\Psi_m^h(\Omega_n^h(s_n^\lambda))) \text{ for } d = m - \gcd(n, m) \quad (16)$$

$$\Delta_d(\Psi_m^t(\Omega_n^h(s_n^\lambda))) \text{ for } \forall(d \ge 0) \quad (17)$$

$$\Delta_d(\Psi_m^h(\Omega_n^t(s_n^\lambda))) \text{ for } d = n + m - \gcd(n, m) - 1 \quad (18)$$

$$\Delta_d(\Psi_m^t(\Omega_n^t(s_n^\lambda))) \text{ for } d = n - \gcd(n, m) \quad (19)$$

Due to lack of space we cannot include the proofs here. We only sketch it for (16) and state, that the same principles can be used to prove (18) and (19). Given a vector signal s_n^λ, the combination of de-vectorization and vectorization will transform s_n^λ into a new vector signal s_m^λ. Suppose we have de-vectorized i vectors of s_n^λ, and vectorized j vectors of s_m^λ, what is the largest minimum difference between in and jm? The answer to that question will tell us for how many samples we have to wait in the worst-case, before we can vectorize vector j+1 of s_m^λ, which equals the delay that is necessary for causality. This question can be reformulated as follows. Given are two number sequences $a_i = in$ and $b_j = jm$. What is the maximum difference of an a_i to the smallest $b_j \ge a_i$? These differences are $d_i = (m - (a_i \bmod m)) \bmod m$. Hence, we have to show that $\max(d_i) = m - \gcd(m, n)$. First we show that $d_i \le m - \gcd(m, n) \forall i \in N$. Then we show that there exist an i such that $d_i = m - \gcd(m, n)$. The outer mod operation in the d_i expression is there, to avoid d_i becoming equal to m. This condition is equivalent to

$$(a_i \bmod m) \ne 0 \quad (20)$$

Thus, we can reformulate the equation as $m - (a_i \bmod m) \le m - \gcd(m, n)$ and, consequently, $a_i \bmod m \ge \gcd(m, n)$. Assuming condition (20), this is correct due to the fact, that $\gcd(m, n)$ divides both, m and a_i and therefore also $a_i \bmod m$ (e.g. see [13] chapter 5).

Next, we have to show that there exists an i such that $d_i = m - \gcd(m, n)$. Under condition (20), we have $in \bmod m = \gcd(m, n)$. Again using the fact that $\gcd(m, n)$ divides both m and in, the equation

$$\frac{in}{\gcd(m, n)} \bmod \frac{m}{\gcd(m, n)} = 1 \quad (21)$$

has always a solution in $i \in N$ for any $m, n \in N$, which concludes the proof.❑

Conditions (16) through (19) can be interpreted in several ways. For modelling they describe the situations, when a process can safely use input data and when not. For instance in figure 4a, process B may not safely use values produced by process A, if its behaviour depends on the order

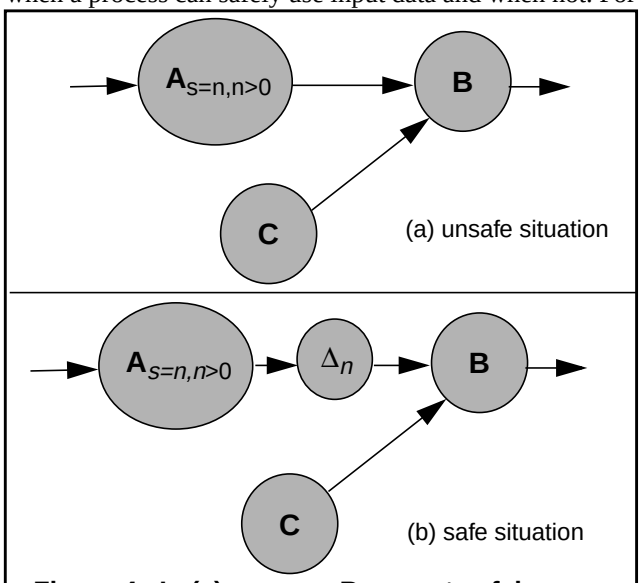

Figure 4. In (a) process B cannot safely use values produced by A; in (b) it can.

of values from *A* and *C*. Because *A* is not causal, a value from *A* could appear earlier in the model than it could in reality, thus it might happen that *B* sees an event $e=(t,v)$ from *A* before an event $e' = (t', v')$ from *C* even though $t > t'$. In figure 4b this is not possible.

Below we discuss other applications of these conditions.

Feedback loops. The theory of metric spaces [6] and the discussion in [18] tell us, that simple causality (equation (5)) is not sufficient to guarantee deterministic behaviour in feedback loops. Rather, a process in a feedback loop must be *delta causal*, i.e. it must comply with the stronger condition

$$d(F(s), F(s')) < \delta d(s, s') \qquad (22)$$

with some strictly positive number δ. This means, the outputs of a process must react to the inputs with a certain delay, which cannot be arbitrarily small. The equations (14) through (19) can be easily reformulated to derive delta causal processes by replacing the delay processes Δ_d by $\Delta_{d+\delta}$ with a fixed, strictly positive number δ. For instance, $\Delta_n(\Psi_n^h)$ is causal but not delta causal; however, $\Delta_{n+\delta}(\Psi_n^h)$ is also delta causal.

This means for modelling, that the outputs of each process in a feedback loop may influence its own inputs only after a strictly positive delay δ, which may not become arbitrarily small.

3. Applications

3.1 Co-modelling of Matlab and SDL

At the system level it is desirable to model and simulate control and data flow parts together. Traditionally, designers have used different and separate simulation environments and languages for these two tasks.

Matlab [22] is an example of a popular language used for the development of data flow and signal processing algorithms. In Matlab programs input data is transformed by applying functions to input parameters. Matlab provides very powerful and convenient operators for vectors and matrices. Matlab does not have a notion of time but uses data dependences to define the order of events.

SDL [11] is based on communicating concurrent state machines. Its execution semantics is a discrete event model. It has been used for a long time to describe and specify control dominated systems, in particular telecom applications.

The composite signal flow model can be used as a basis for co-modelling and co-simulation of systems with these two languages. This has been described in detail in [2], which also includes details of the implementation and gives elaborate examples to motivate and illustrate this modelling method.

SDL is used to describe the system structure and the connectivity between processes, as well as processes, that operate solely on non-periodic signals. Processes, that operate on sampled and vectorized sampled signals are modelled as Matlab functions. In figure 5 *A* and *B* are SDL processes, *C* and *D* are Matlab functions, *a*, *b*, *c*, *d* and *e* are non-periodic signals, and f^λ, g^λ and h^λ are sampled signals.

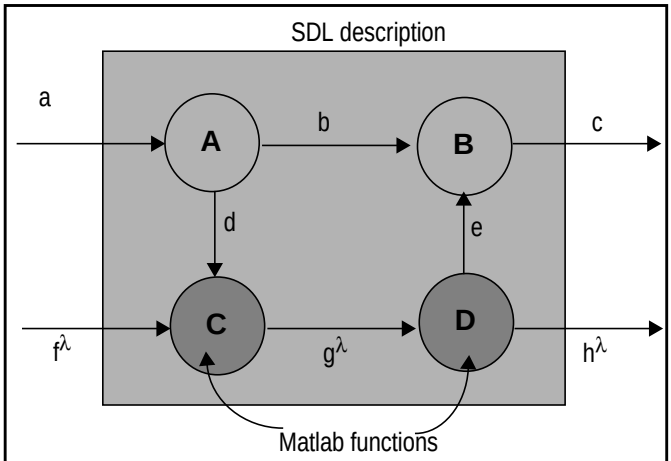

Figure 5. The composite signal model implemented in SDL and Matlab.

Since in Matlab it is both convenient and efficient to utilize the powerful vector and matrix operations, most Matlab functions operate on vectorized, sampled signals. The vectorization is implicitly performed on the boundaries between the SDL environment and the Matlab functions. [2] gives modelling conditions which are less general but consistent with equations (16) - (19).

[2] also uses Pin_E, Pin_D and Punch processes to synchronize and communicate between data flow and control parts. However, they are not used as explicit processes but as elementary communication mechanisms built-into the modelling method and implementation. In addition, [2] describes a *Bucket event*, which is similar to a Punch process but differs in that it takes a non-periodic signal also as its first input, rather than a sampled signal.

[2] is a good illustration, how the composite signal flow model can be implemented. In fact, the composite signal flow model has been developed as one result of the effort described in [2], based on the insight, that the developed concepts are much more general and applicable to other language combinations.

3.2 Timing analysis

If the processing is only an artifact of modelling for the sake of efficiency or convenience, equations (16) through (19) do not reflect timing behaviour in any way. But if the transformations of vectors resemble data dependences in the implementation, equations (16) through (19) constitute in fact timing constraints (see for instance [3] section II.A). Consider a function F which transforms a vector of size n

into a vector of size m, $F : W_n \rightarrow W_m$. Assume that F needs all the values of the input vector to produce any of the values of the output vector, e.g. a compression function, which operates on words of a given length represented by the input vectors. We can model F with a process, which operates on vectorized sampled signals. Since the primary inputs and outputs of our system are not-vectorized sampled signals, we have to include the vectorization and de-vectorization processes in our model (figure 6). Since the

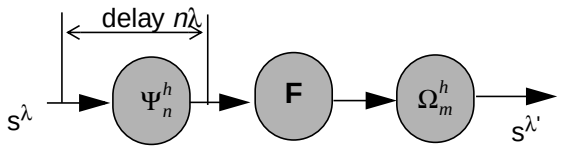

Figure 6. Delay of the vectorization process interpreted as timing condition.

head vectorization process Ψ_n^h is non-causal with severity level n, we need to combine it with a delay process to make the resulting process causal. In the case of figure 6, we do not need the delay process for simulation, because process F does not depend on any other process. However, a timing analyser can infer the necessity of a delay process and establish a timing condition on the vectorization process, stating that the process will take at least $n\lambda$ time units. Note, that the vectorization process might not exist separately in the implementation; it might be part of process *F*. In this case the timing condition would apply to process *F*. Hence, the timing analyser will perform better with more information about the relation between model and implementation. The minimum assumptions are the causality requirement, and that the vectorization is indeed somewhere implemented.

Note, that we derive an equivalent timing condition if we use tail vectorization and tail de-vectorization in figure 6, but not necessarily on the same process. Because tail vectorization is causal, but tail de-vectorization is not, the time constraint has to be placed on the latter.

Consider another example with two functions F and G. F operates on n-sized vectors and G on m-sized vectors (figure 7). Causality requires two delay processes, one for the head

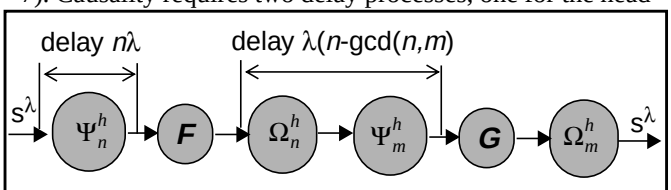

Figure 7. Causality induced delays for two functions operating on different vector sizes.

vectorization at the left and one for the de-vectorization-vectorization combination in the middle. Again, these can be interpreted as timing constraints on the processes, which include the corresponding signal transformations.

Note, that this analysis can be combined with rate analysis of the sampled signals, to further refine the timing requirements of the system.

Feedback loops. What has been said about modelling of feedback loops in section 2.5, is also valid for the derivation of timing constraints. To ensure delta causality of processes within feedback loops, stronger variations of the equations (14) through (19) must be used. This is necessary to avoid oscillations or non-deterministic behaviour in the implementation.

3.3 Parallel simulation

Systems, modelled as parallel processes, exhibit a natural parallelism, that can be exploited for parallel simulation. However, the potential to speed up simulation seems to be fundamentally constraint by the model of computation used. Discrete event models are inherently difficult to simulate efficiently in parallel due to the synchronizing global event queue [9].

A system model based on composite signal flow can be partitioned for parallel distributed simulation, if the following two conditions are met.

(A) A partition must be a causal process. However, not all the constituent processes within the partition need to be causal.

(B) No non-periodic signal crosses a partition boundary, i.e. only sampled signals can be used for communication between different partitions.

Both conditions serve the purpose that a receiving process must be able to know, how many values to expect from other partitions, before proceeding with its own computation. The causality condition enforces, among others, that a process waits for all the data needed, before it computes a result (figure 8). For non-periodic signals it is not possible

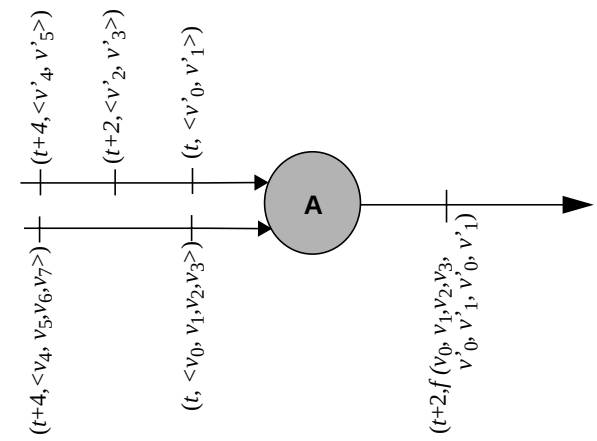

Figure 8. Process *A* must wait until *t*+2 if it uses values v'_0 through v'_3 for a particular computation *f*.

to know, when the next event will occur.

Non-periodic signals can be replaced by functionally equivalent sampled signals, by transmitting an empty value periodically. In this way, timing and synchronization information would be encoded into the signal.

4. Conclusions

Heterogeneous modelling will become more important due to increasing complexity and heterogeneity of electronic systems. Due to various historical and fundamental reasons, different parts of a system will be modelled with different languages. This results often in the challenge to reconcile very different modelling concepts in a sound way. We have addressed this problem for event based models on one hand and vectorized signal processing models on the other hand. By providing a unifying model of computation, the composite signal flow, the two different modelling worlds can be combined. We have applied this concept to integrate two very popular but profoundly different languages, SDL and Matlab. We have also shown that the concept can be applied to other problems as well, such as timing analysis and parallel simulation.

5. References

[1] T. Benner, J. Henkel, and R. Ernst, "Internal representation of Embedded Hardware-/Software Systems", *2nd IFIP International Workshop on Hardware-Software Co-Design*, May 1993.

[2] Per Bjuréus and Axel Jantsch, "Heterogeneous System-level Cosimulation with SDL Matlab", P*roceedings of the Forum on Design Languages* (FDL), 1999.

[3] I. Bolsens, H. de Man, B. Lin, K. van Rompaey, S. Vercauteren, and D. Verkest, "Hardware/Software Codesign of Digital Telecommunication Systems", *Proc. of the IEEE*, March 1997.

[4] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", *International Journal of Computer Simulation*, 1992.

[5] P. Coste, F. Hessel, P.L. Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, and A.A. Jerraya, "Multilanguage Design of Heterogeneous Systems", *Proc. of the 7th International Workshop on Hardware/Software Codesign*, pp. 54-58, May 1999.

[6] V. Bryant, *Metric Spaces*, Cambridge University Press, 1996.

[7] J.-M. Daveau, G. F. Marchioro, C. A. Valderrama, and A. A. Jerraya, "VHDL generation from SDL specifications", *Proceedings of Computer Hardware Description Languages*, April 1997.

[8] W. Ecker, "Using VHDL for HW/SW Co-Specification", pp. 500-505, *European Design Automation Conference*, September 1993.

[9] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models, Validation and Synthesis", *Proceedings of the IEEE*, March 1997.

[10] Petru Eles, K. Kuchcinski, Zebo Peng, and A. Doboli, "Hardware/software partitioning of VHDL system specifications", *European Design Automation Conference (Euro-DAC)*, 1996.

[11] J. Ellsberger, D. Hogrefe, and A. Sarma, S*DL - Formal Object Oriented Language for Communicating Systems*, Prentice Hall, 1997.

[12] R. Ernst and J. Henkel, "Hardware-Software Codesign of Embedded Controllers Based on Hardware Extraction", *Proc. International Workshop on Hardware-Software Co-Design*, 1992.

[13] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, *Oxford Science Publications*, fifth edition, 1979.

[14] Rachid Helaihel and Kunle Olukotum, "Java as a Specification for Hardware-Software Systems", *Proceedings of the International Conference on Computer-Aided Design*, 1997.

[15] A. A. Jerraya and K. O'Brien, "SOLAR: An Intermediate Format for System-Level Modeling and Synthesis", *Codesign: Computer-aided Software/Hardware Engineering*, IEEE Press, edited by J. Rozenblit and K. Buchenrieder, chapter 7, 1995.

[16] E.A. Lee and D.G. Messerschmitt, "Synchronous Data Flow", *Proceedings of the IEEE*, pp. 1235-1245, September 1987.

[17] E. A. Lee and T. M. Parks, "Dataflow Process Networks", *Proceedings of the IEEE*, May 1995.

[18] E.A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 12, pp. 1217-1229, December 1998.

[19] C. Liem, F. Nacabal, C. Valderrama, P. Paulin, and A. Jerraya, "System-on-a-Chip Cosimulation and Compilation", *IEEE Design and Test of Computers*, pp. 16-25, April-June 1997.

[20] Bill Lin, "A System Design Methodology for Software/Hardware Co-Development of Telecommunication Network Applications", *Proceedings of the Design Automation Conference*, 1996.

[21] P. Le Marrec, C. A. Valderrama, F. Hessel, A. A. Jerraya, M. Attia, and O. Cayrol, "Hardware, Software and Mechanical Cosimulation for Automotive Applications", *Proc. of the Ninth International Workshop on Rapid System Prototyping*, 1998.

[22] *MATLAB: High-performance Numeric Computation and Visualization Software*. User's Guide, 1992.

[23] S. Narayan, F. Vahid, and D. D. Gajski, "System Specification with SpecCharts Language", *IEEE Design & Test of Computers*, December 1992.

[24] C. Passerone, J. Martin, R. Passerone, L. Lavagno, C. Sansoe, R. McGeer, and A. Sangiovanni-Vincentelli, "Modeling Reactive Systems in Java", *ACM Transactions on Design Automation of Electronic Systems*, vol. 3, no. 4, October 1998.

[25] A. Poigne, M. Morley, O. Maffeis, L. Holenderski, and R. Budde, "The Synchronous Approach to Designing Reactive Systems", *Formal Methods in System Design*, Kluwer Academic Publisher, vol. 12, no. 2, pp. 163-188, March 1998.

[26] B. Svantesson, S. Kumar, A. Hemani, "A Methodology and Algorithms for efficient interprocess communication synthesis from system description in SDL", *Proc. of the IEEE International Conference on VLSI Design*, 1998.

[27] J.S. Young, J. MacDonald, M. Shilamn, A. Tabbara, P. Hilflinger, and R. Newton, "Design and Specification of Embedded Systems in Java Using Successive, Formal Refinement", *Proceedings of the 35th Design Automation Conference*, 1998.