

# A Component Model for Synchronous VLSI System Design

Dominique Lavenier, Roderick McConnell

**N° 2285**

mai 1994

PROGRAMME 1

Architectures parallèles,  
bases de données,  
réseaux et systèmes distribués



*rapport  
de recherche*



# A Component Model for Synchronous VLSI System Design\*

Dominique Lavenier, Roderick McConnell\*\*

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués  
Projet API

Rapport de recherche n° 2285 — mai 1994 — 26 pages

**Abstract:** We present a model for a library of synchronous VLSI components, which describes both their function and their timing diagrams, without describing the internal implementation of the circuit. Our model is based on Synchronous Data Flow, with extensions to model VLSI circuits, and to verify that a system designed with VLSI components from our library is properly synchronized. Initialization and termination conditions are explicitly checked.

**Key-words:** digital signal processing, specialized systems design, VLSI

*(Résumé : tsvp)*

An abbreviated version will be presented at RSP-94

\*This work was partially funded by the French Coordinated Research Program ANM of the French Ministry of Research and Space, by the Esprit BRA project No 6632 NANA-2, and by the Doctoral-candidate Network for System and Machine Architecture of the DRED.

\*\*{Dominique.Lavenier}{Roderick.McConnell}@irisa.fr

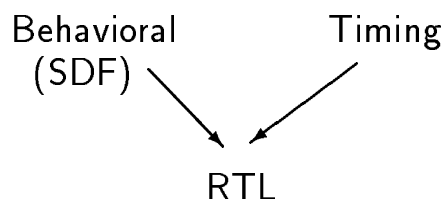
# Un modèle de composant pour la conception des systèmes VLSI synchrones

**Résumé :** Nous présentons un modèle de bibliothèque de composants VLSI synchrones périodiques qui précise à la fois le fonctionnement et l'aspect temporel des composants, sans détailler la réalisation interne. Ceci permet de s'assurer qu'un schéma dessiné à l'aide de ces composants est bien synchronisé. Les conditions d'initialisation et de terminaison sont aussi considérées. Nous introduisons une boîte-à-outils logicielle pour préciser les horloges, puis nous les appliquons à l'élaboration des modèles RTL à partir des modèles comportementaux.

**Mots-clé :** conception de systèmes spécialisés, VLSI, traitement digital de signaux

# 1 Introduction

The transition between a behavioral model of a system and a Register Transfer Level (RTL) model is often a difficult step, involving a complete re-write of each component. In this article we propose a systematic approach to developing an RT-level system model, starting from the behavioral level. The approach is based on a Synchronous Data Flow (SDF) design methodology for transitioning between a behavioral model and synchronous VLSI circuits. We consider an RTL simulation component to be composed of two sub-elements (as shown below): a timing aspect which determines the moments at which the inputs and outputs exist; and an operational aspect which performs the actual calculation, i.e. the behavior. We have chosen this composition because it permits us to preserve the behavioral aspect of components, while adding the necessary timing for RTL simulation. In order to develop an RTL model, we require detailed timing information for the VLSI circuits (such as that given on a datasheet) but not the specifics of the internal implementation, making it practical to insert models of commercially available circuits.



Initialization and termination sequences are an important part of an implementation, particularly when working with complex VLSI circuits. On the other hand, when working with a behavioral model, these timing considerations either are ignored or are expressed differently. We model the hardware initialization sequence using the timing sub-element of the component.

Our component model depends heavily on two properties, one from Synchronous Data Flow (SDF) and the other from synchronous circuit design. In particular, it is assumed that all operations are periodic (as in SDF [21][22]) and that the inputs and outputs of a component can be calculated based on a single clock, as with synchronous circuits [28]. In other words, we assume that each component performs a certain cyclic operation independent of the data, and that the timing of input and output events can be calculated relative to this cyclic operation using a single clock. A more formal discussion of these constraints is given in [14],[15].

The two-sub-element components, together with a *toolbox* of useful timing subroutines, offer several practical advantages. The components need not be described internally at a gate level. And the separation of functions make it easy to change the “technology” of the simulation library. Components for different purposes, e.g. circuit-level design or formal verification, are easily developed once the timing toolbox has been expressed in a suitable technology. The model also adapts well to the typical stages in the design process, beginning at a block-level and decomposing down to circuits. We demonstrate this utility with an example, a change between block-level and circuit-level design.

The domain of real-time digital signal processing has inspired numerous efforts to aid designers, certainly more than can be presented here; we restrict ourselves to a few which have directly influenced our work. We cite Ptolemy [6], which implements Synchronous Data Flow graphs directly, and SYNDEX [11] [17], a multi-processor environment based on the SIGNAL [18] synchronous language, as two systems for developing signal-processing applications based on the use of programmable components such as DSP processors. Likewise, for synchronous circuit design there exist two (related) systems, Cathedral-IV [27] and the Phideo system [30], which use a component model that includes hardware-level timings; these two use Silage [12] for algorithm description. We emphasize that this is an arbitrary selection, as even considering just the domain of high-throughput synchronous system design, there are many models and environments.

Multi-language simulations as well are nothing new. Many designers combine timing and behavioral simulation using VHDL together with C, or use similar combinations of proprietary languages. However, to our knowledge these combinations are ad-hoc, and do not benefit from a formal SDF-based framework.

After a brief discussion of the advantages of SDF, and its limitations when attempting to apply it to VLSI circuits, we present our component model. We then present several families of libraries based on this component model. A toolbox of subroutines which are helpful for building the timing sub-element [25] is introduced, then applied to the construction of a simple system. We finish by presenting an example, components for motion video image compression.

## 2 Synchronous Data Flow

The Synchronous Data Flow (SDF) graph offers an elegant block-level data-flow representation for certain real-time Digital Signal Processing (DSP) applications. SDF is a variant of the traditional data flow graph [1][8] where the amount of data consumed and produced by each node for all its inputs and outputs, is fixed in advance. Data flow graphs are often used in signal processing to model real-time applications, because they offer an intuitive, visually-oriented approach [7].

An application expressed as an SDF graph has a priori been decomposed into manageable sub-problems, each performing a distinct operation; at the same time, the unity of an operation or of a block of data is preserved. This provides modularity and the possibility

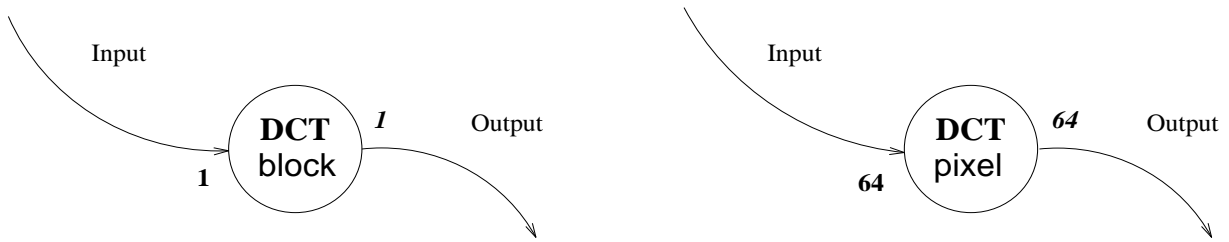


Figure 1: SDF node at block and pixel level

of describing an application at different levels of abstraction. Hence a designer works in a more natural and intuitive setting. RTL timing considerations do not enter into an SDF description - operations are described purely in terms of their behavior, i.e. in terms of the inputs *consumed* and outputs *produced*. We consider our behavioral model to be an SDF graph.

An SDF graph can be analyzed to establish good behavior [19]. In particular, a transfer matrix can be formed from the inputs and outputs of all nodes, with which it can be established that the input and output buffers will be bounded. One can also analyze the graph for loops without an intervening delay, which would imply a deadlock during execution.

In the remainder of this section, we present the SDF model in greater detail, followed by an analysis of its limitations, and an introduction to our approach for applying SDF concepts to the design of synchronous hardware.

## 2.1 The SDF model

In the SDF model, a signal processing application is described in terms of *static* transfers and operations. The number of *consume* elements needed for an operation to commence, and the number of *produce* elements resulting from an operation, are fixed at design time. Furthermore, the nature of these produce and consume elements is not fixed - their granularity can vary from single words or pixels, to blocks of arbitrary size, representing for example entire images [20]. The consume elements are buffered at the input to an operator until an input threshold is reached; the operator is then *fired*, or started. In the abstract model, the results are then available immediately, i.e. the produce elements are produced with no delay.

The number of consume elements needed to launch an operation, depends on the operation and on the granularity of the elements. For example, a two-dimensional discrete cosine transform (DCT) which requires a block of 8x8 pixels as input, may be represented in coarse-grain data flow as consuming 1 block to start processing. Or, at a more fine grain, the DCT may be represented as consuming 64 pixels before starting. The first representation assures that all the inputs will arrive at the same instant, while the second representation implies a buffering of input elements until all have been received. In Figure 1 an SDF node for a DCT is given at a block and at a pixel level.

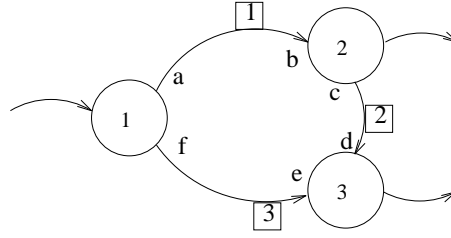


Figure 2: A generic three node dataflow graph

### 2.1.1 Synchronous dataflow graphs

An SDF graph is a collection of connected nodes. Each node represents a specific operation, the complexity of which depends on the level of description of the associated functional block. The operator will operate on signals which contain an infinite stream of values. Operations must also be independent, i.e. all interactions between operators must be specified by means of the static transfers. This yields a graph in which all operations are nodes, and all transfers are arcs between nodes. The signal-flow graphs presented by Lee and Messerschmitt are called *synchronous* to express the fact that an operator is invoked when a pre-specified number of new values are available at all inputs. We emphasize that the threshold for each input, as well as the number of inputs consumed and the number of outputs produced, is fixed in advance as shown in Figure 1.

### 2.1.2 Consistency

When developing a synchronous data flow application, it is important that the SDF graph has a bounded cyclic schedule. This in turn implies that each node can be invoked a bounded integer number of times, after which input and output buffers will return to a previous state of “fullness”. For example, if we have two nodes with a communication dependency between them, we can pose the question whether it is possible to call each node a finite number of times, allowing a complete exchange of the specified number of data, without accumulation or starvation on the part of either node. The property of existence of a bounded cyclic schedule is called *consistency* in [19]. Lee & Messerschmitt provide conditions for consistency, based on a transfer matrix which represents the SDF graph.

### Validity conditions for consistency

For a given SDF graph, we would like to find a periodic schedule  $\vec{r}$ . If such a schedule exists, we can solve a system such as the one given in Figure 2. The number of elements produced and consumed for each link is represented by the expressions:

$$\left\{ \begin{array}{l} a * r_1 = b * r_2 \\ f * r_1 = e * r_3 \\ c * r_2 = d * r_3 \end{array} \right\}$$

$$\begin{pmatrix} a & -b & 0 \\ 0 & c & -d \\ f & 0 & -e \end{pmatrix}$$

Figure 3: Topology Matrix associated with Figure 2

where  $r_i$  represents the number of times that node  $i$  is “fired”. We can then verify the equality between elements produced and elements consumed:

$$\left\{ \begin{array}{l} a * r_1 - b * r_2 = 0 \\ f * r_1 - e * r_3 = 0 \\ c * r_2 - d * r_3 = 0 \end{array} \right\}$$

We can associate a *topology matrix*  $\Gamma$  as shown in Figure 3 with the graph given in Figure 2. A column is associated with each node, and a row with each edge. The edges have been labeled in an arbitrary fashion, as shown in the little boxes in the Figure 2. The  $(i, j)^{th}$  entry defines the number of data exchanged between node  $i$  and  $j$ . If a node  $j$  consumes data on edge  $i$  the value is negative; otherwise it is positive, or equal to zero if edge  $j$  is not connected to node  $i$ .

Lee and Messerschmitt show that a necessary condition for the existence of a valid solution schedule, is that  $rank(\Gamma) = s - 1$ , where matrix  $\Gamma$  is the matrix of outputs and inputs between the different nodes of one graph, and  $s$  its number of nodes [21] [22]. In fact, if a solution  $\vec{r}$  exists, it will be a null vector for the transfer matrix  $\Gamma$ :

$$\Gamma \vec{r} = \vec{0}.$$

Likewise any multiple of this null vector will also be a solution, yielding a family of solutions. These solutions correspond to a single or multiple iterations of the system period. Intuitively, one can see that there must exist a single family of solution vectors, because the system must be able to run for one or more periods, while at the same time, the absence of conditional communications implies that there is only one possible execution scheme. The system itself may be a subsystem of a larger graph, in which case the minimum period of the larger graph will be a multiple of the periods of all subgraphs. This can be seen intuitively from the fact that if  $\Gamma_i$  and  $\Gamma_j$  are two sub-matrices representing subsystems of  $\Gamma$ , if they share a link, they will share a period which must satisfy both null vectors.

In order to render the calculations more manageable, we will generally consider only the least of the solution vectors of the system (or subsystem) under consideration.

## 2.2 Limitations of SDF

The SDF representation has limitations, the most fundamental being that it can only be applied to a limited sub-class of signal processing applications, namely those where the data

transfers can be statically specified. In addition, breaking the complete application into sub-problems, as represented by the nodes of an SDF graph, may limit the possibilities for global optimization. Approaches such as that of PHIDEO [30], however, allow automated optimization of systems which fit the Synchronous Data Flow model, particularly in terms of memory allocation [31].

Given a suitable application, there is another limitation which to us appears relevant: SDF as presented by Lee & Messerschmitt, and implemented in the SDF domain of the design system PTOLEMY [6], appears better adapted to modeling software than hardware. This is due to the block nature of the transfers and operations, which correspond closely to an input or output buffer and a subroutine call, respectively. While buffers and subroutines are well-suited for implementation using one or more programmable DSP's, they are poorly suited to a dedicated VLSI hardware implementation.

It is clear that there can be a great deal of similarity between an SDF graph and an RTL model of a hardware-based signal processing system. In an application such as motion video compression in real time, complex operations such as a discrete cosine transform (DCT) are often performed by a single dedicated circuit, which takes a block of data at the input and produces a block of data at the output. The principal difference for the designer is in the timing of data transfers. Produce and Consume thresholds in SDF must be replaced by a stream of data, word by word, to reflect the operation of the hardware at a Register Transfer level. It is basically a bandwidth and storage problem - there aren't enough pins on a chip to pass blocks of data at a time, nor does one want to store data unnecessarily.

### **2.3 Approaches for Hardware**

One approach to modeling a hardware system at the Register Transfer level, is to break down block-oriented operations into simpler sub-operations, until each operation works on a single word of data. We discard this approach, as it yields an unwieldy and counterintuitive graph, and is also a poor model for VLSI circuits.

We choose instead to model blocks of data at the RT level as being transferred serially, word by word. Nodes in our graph, and components in our library, are composed of two sub-elements, a timing component and an operator. We use the timing component to represent the timing of a cyclic transfer, whether it be at a block level (native SDF) or serial stream (circuit-level).

In order to make the change-of-model easier, our library of elements contains both block-level and circuit-level timing representations for different operations. We note that typically the timing sub-element changes, while internally the operation remains an operator on blocks of data, the same as would be used in PTOLEMY. In the following section, we describe our RTL component model.

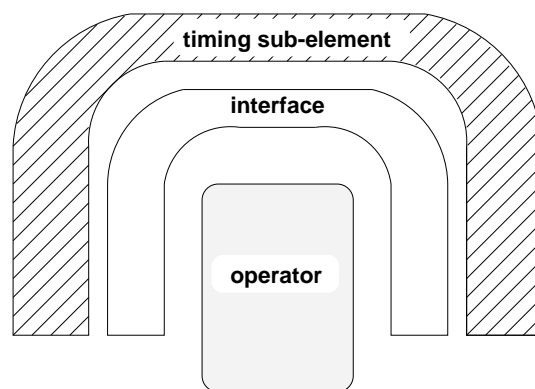


Figure 4: A Three-part Library Component

### 3 RTL Components

Our approach specifies RTL components using a synchronous timing sub-element. We completely separate timing aspects of a component from operational aspects, reflecting the usual conceptual distinction between the *function* of a circuit and the *timing* of a circuit. The timing sub-element is used to transition between an SDF graph and an RTL circuit model. The operation sub-element which performs the function is treated as a “black box”, without regard for its implementation. This approach differs from existing systems for developing periodic circuits, such as CATHEDRAL [27], PHIDEO [30], or GAUT [24], which use knowledge of the internal operations to perform allocation and optimization. We emphasize that our approach is targeted at system-level RTL design, and not at individual component design.

#### 3.1 A Three-part Maquette

We use a three-part maquette as the starting point for each component. The maquette includes the timing specification, the operation, and interface “glue” between the input/output and the operation (Figure 4). Note that not all parts of the maquette need be filled - for example, there is rarely any input interface required if both input and operation occur at a block level.

Below we describe briefly each of the three parts in Figure 4. The part labeled **operator** is an arithmetic or logical operator which describes the behavior of the component. In PTOLEMY this would be called an *actor*. Typically, this is a C subroutine. The operator takes input data in an arbitrary format (say, a block of 8 x 8 words), and yields results without any delay, corresponding to a subroutine call. The *timing diagram* of the component is presented to the exterior by the **timing sub-element**. The timing includes a check on all inputs that they indeed arrive when the timing diagram indicates that they are expected, and explicit generation and suppression of outputs, also according to the timing diagram. An **interface** is used to handle any differences in format between the timing sub-element

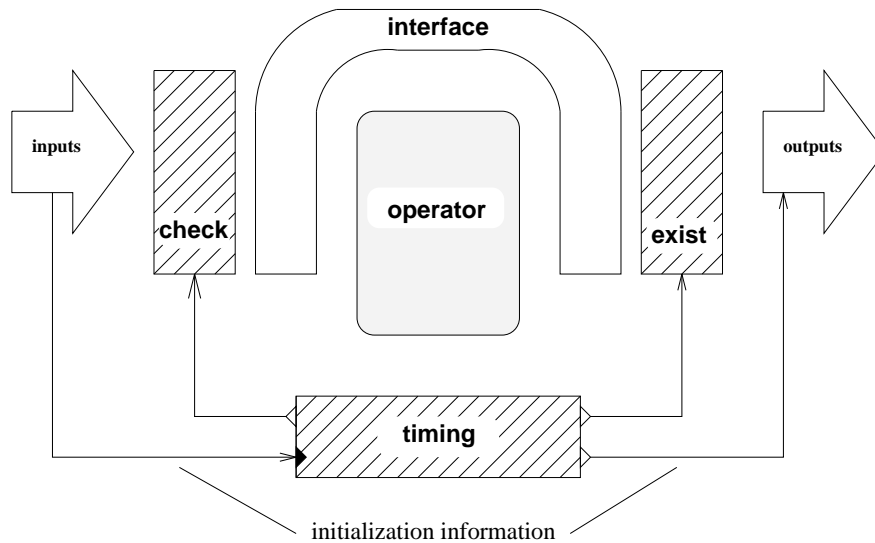


Figure 5: Detail of the Timing sub-element

and the operation; this interface, for example, handles the buffering of a stream of words for a block-oriented operator.

## 3.2 The Timing Sub-element

The timing sub-element represents a novel approach to RTL descriptions, and we present it here in greater detail. The timing sub-element includes three distinct functions, as indicated in Figure 5: calculating the timing diagram of a component; checking that the inputs indeed arrive when they should; and controlling the existence of the output values so that they only “exist” when the timing diagram specifies that they are valid (control of the existence of the outputs permits checking the inputs at the following component).

### 3.2.1 Initialization and Termination

RT-level initialization and termination conditions of a component, as specified in the timing diagram of a circuit datasheet, are handled by the timing sub-element. This includes reset sequences and termination (e.g. emptying buffers). The initialization and termination information is indicated in Figure 4 by the direct connections between the **timing** sub-element and the input & output. These connections carry initialization information between the timing sub-elements of a system. Because the behavior of an operation is not affected by the initialization, this information is not passed to the **operator**.

### 3.2.2 Timing Parameters

The timing sub-element is developed from the software tools described in [25]. Here we will just give a brief summary of the parameters which specify the timing. These parameters

will be used in the RTL description of a system in the following section. Parameters to the timing tools are: the `LATENCY` of the component; the number of input events per period; the number of output events per period; and the number of clock cycles in a `PERIOD`. The `LATENCY` of the component is specified in order to handle initialization and termination: the output is not considered valid until `LATENCY` clock cycles after the input arrives, and likewise the output continues to be valid until `LATENCY` cycles after the input is no longer valid. All components are periodic, as demanded by SDF; the `PERIOD` is the number of cycles of the system clock per period of the given component.

### 3.2.3 Implementation

The timing sub-element must be written in a language capable of expressing the moments when a signal exists in the RTL sense; we use `SIGNAL` [2] or VHDL [23]. The behavior, or operation, on the other hand can be written in whatever language is most appropriate; we typically use C [16]. In this article we will present components using `SIGNAL` (a synchronous dataflow language) for the timing sub-element, and C for the operation sub-element. Differences in the representation of data between the two sub-elements often creates the need for an interface, to reorganize data passing between the timing sub-element and the operational sub-element; we implement this interface using languages from the two sub-elements as appropriate.

We note that typically the operation, the function of a component, remains unchanged regardless of the level at which we regard the timing. At a block level, the timing sub-element is trivial, but the operation sub-element must be fully developed, in order to support simulation. At a circuit-specific level, the timing element must reflect the timing diagram of a particular circuit, perhaps including an initialization and/or termination sequence.

Changes are required in the interface as well, as the timing changes, to properly organize the data. Typically a generic interface serves to collect a stream of data into an input buffer, and to send a stream of data from an output buffer. For example, if the timing diagram specifies a stream of words and a certain latency between the first input and the first output, and the operator is a C subroutine which operates on a block of words, an interface is used to buffer the input stream and store the output block until it should be transmitted.

## 4 The Timing Toolbox and SDF

In this section we summarize two of the tools which have proven useful for designing the timing sub-element and interface logic, and explain how they relate to a Synchronous Data Flow graph. For a more detailed explanation of the tools, the reader is referred to [25]. The tools presented here will be used in an example of system-level synchronization in a later section.

### 4.1 The H Process

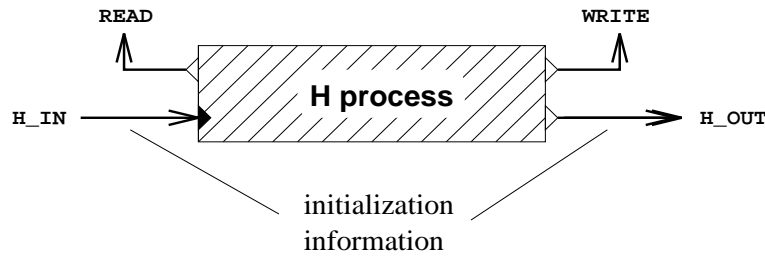
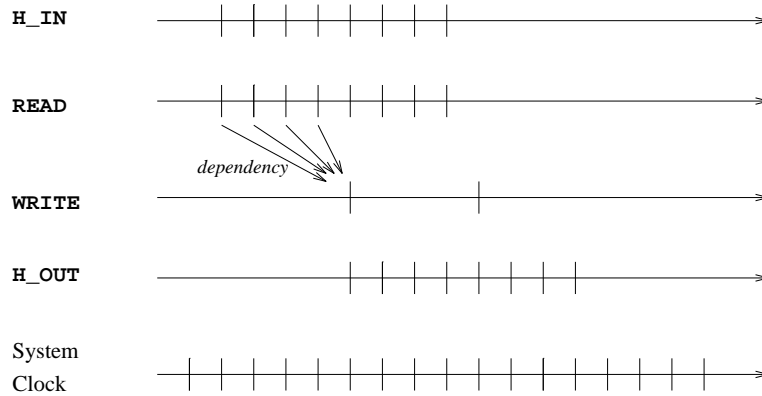


Figure 6: The H Process

Figure 7: Events associated with  $H(4,4,1,4)$ 

The H process shown in Figure 6 is the simplest of the timing processes. It contains two counters, to count input and output events, and a delay line for the initiation and termination considerations. In the case of some SDF nodes, with one block of input and one block of output, this timing tool is sufficient to describe the synchronization of a node; otherwise, other tools such as the R process are needed.

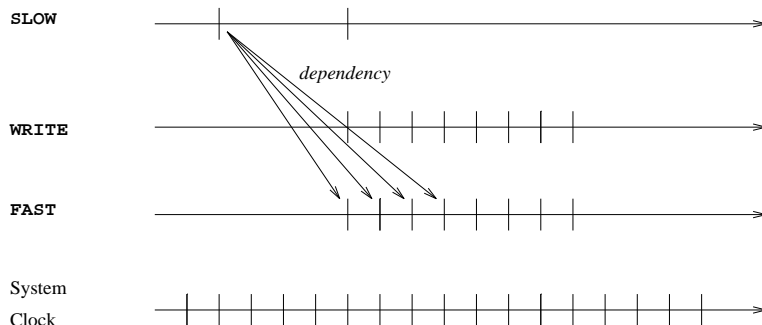
All signals (denoted by arrows in Figure 6) are boolean, and synchronous with a common clock. Whenever READ is true, there is an input event, and whenever WRITE is true, there is an output event. Input processing starts when H\_IN becomes true, and finishes when H\_IN is false. The signal H\_OUT goes true when the output starts being valid, i.e. after the latency of the component has passed. Parameters to this process are: the LATENCY between input and output; the number of input events RCLK per period; the number of output events WCLK per period; and the number of clock cycles in a PERIOD. In Figure 7, we show the instants at which the outputs are true, given an H process  $H(4,4,1,4)$ .

## 4.2 The R Process

The R process shown in Figure 8 is used to resynchronize values, the synchronization of which depends on the system, rather than just a single component. It is used together with the interface logic, to buffer and up-sample values. It is particularly useful for control



Figure 8: The R Process

Figure 9: Events associated with  $R(4, 1, 4, 4)$ 

values which arrive less frequently than data, but must nevertheless be present (and properly synchronized) each time the operation subroutine is called.

As with the H process, all signals are synchronous with a common clock. The incoming values arrive at the input SLOW. They are transferred to the output FAST whenever WRITE is true, after a specified latency. This process is also periodic, to be used together with periodic components. Parameters to this process are the same as those for the H process: the LATENCY between input and output; the number of input events RCLK per period; the number of output events WCLK per period; and the number of clock cycles in a PERIOD. In Figure 9, we show the instants at which the outputs are true, given an R process  $R(4, 1, 4, 4)$ .

## 5 Applications of our Component Model

Our abstract model leads to a variety of application-specific (or simulation-specific) implementations, depending on the desired results. As mentioned earlier, different sub-elements are written in a variety of languages: C to simulate complex VLSI functional blocks; SIGNAL to allow formal verification; and VHDL to permit synthesis of control structures for an implementation. It is essential for checking the synchronization that the timing sub-element be written in a language capable of representing temporal constraints. The operation sub-element, on the other hand, can be written in whatever language best expresses the (abstract) operation to be performed.

In this section, we present three different “technologies” for implementing components, leading to three different uses of the library. The libraries share the same component model, the same abstract operations, and the same parameters. It is also interesting to note that

all three libraries share the same operation sub-elements, written in C which makes for reasonably fast simulations.

A simple library of components written with SIGNAL and C allows a functional simulation of a system at both an abstract and a component-specific level. This permits the designer to experiment with a given configuration, as in PTOLEMY and to then descend to a circuit-specific simulation without changing the configuration. We present the specifics of such a library in the first of the following subsections.

A library of components with timing sub-elements written using the SIGNAL subset of SIGNAL allows the use of a formal verification tool to verify the proper synchronization of a system. The verification is performed using Ternary Decision Diagrams [9], which are similar to Binary Decision Diagrams [5]. This is presented in the second subsection.

A library of components written using VHDL offers two advantages: the first is the increased possibilities for exchange using this lingua franca of hardware design; the second is the possibility of eventual automatic synthesis of interface “glue” logic from the simulation model. The VHDL-based library is presented in the third subsection.

## 5.1 A Basic Simulation Library

In this subsection we present a basic library of components, which permits one to perform simulations of synchronous data-flow systems. It also supports the evolution of a design, starting with a block-level model as in the traditional Synchronous Data Flow, and moving towards a system composed of circuit models. This basic library offers the advantages typical to a high-level language: information hiding (in this case the specifics of how each operation is implemented); modularity, imposed by the component model; and maintainability, as timing diagrams and functionality are independent of the circuit implementation, and can be updated separately as components evolve.

### 5.1.1 The Block-level Model

The block-level model is inspired by the SDF domain of Ptolemy, and by the block diagrams which seem intuitive to most designers. The model makes it easy to design a trial system, as the components are close to the abstract operations which they implement. The model also makes for reasonable simulation times, as all mathematical and control operations are performed by subroutine calls working directly on blocks of data.

In a block-level component, the timing sub-element is typically trivial, consisting of a specification that all inputs arrive at the same time, and the result is available after a unit delay. The operation sub-element, on the other hand, must be fully developed in order to perform a simulation. Interface “glue” logic is typically limited to a buffer used to store the result of the operation for a unit delay. An example of a block-level component is given in the following section.

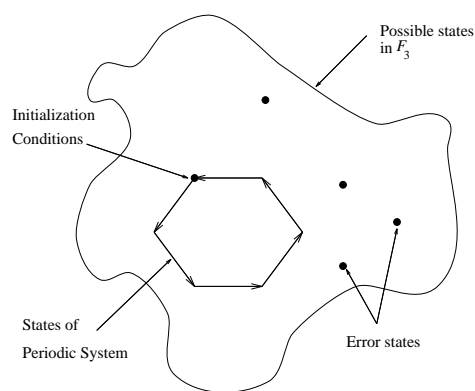


Figure 10: Formal validation of synchronization

### 5.1.2 The Circuit-level Model

The circuit-level component model represents an operation as it would be implemented in a VLSI circuit. This model includes the details necessary for modeling a realistic system built of VLSI circuits, such as initialization phases, delays, etc. Transfers between components are now typically serial, a stream of words, pixels, etc. The timing sub-element must now be fully developed in order to describe the particularities of a certain piece of synchronous hardware. However, the operation sub-element in a circuit-level model remains unchanged from a block-level model. It is typically still a block-oriented subroutine call, which helps maintain a reasonable level of performance for simulations. An interface is usually required between the serial inputs and outputs of the component, and the block-oriented operation sub-element. An example of a circuit-level component is given in a following section.

## 5.2 A Formal Verification Library

It is possible to formally verify the proper synchronization of a system, using a library of components with certain characteristics. The verification is limited to the temporal correctness of transfers between components, i.e. we verify that the time at which a word of data arrives at a component are indeed those instants at which the data is expected. In other words, we verify the synchronization which would be performed dynamically in a self-timed system. Note that this does not in any way guarantee that a system will produce correct results - just that data will not be lost in transit between components.

In order to perform this formal verification, the timing sub-element of each component must be expressed using a subset of SIGNAL limited to boolean values plus the *absence* of value. In mathematical terms, this can be expressed as the algebraic domain of  $\mathcal{F}_3$  [3]. A boolean signal  $X$  would be expressed as  $X \in \{-1, 0, 1\}$ , where  $-1$  represents *false*,  $1$  *true*, and  $0$  the absence of a boolean value for  $X$ .

A Ternary Decision Diagram (TDD) can then be constructed, to which the evaluation tool SIGALI [9] can be applied. Verification consists of evaluating all possible intersections

between nodes in the TDD accessible from the initialization values, and nodes in the TDD which represent error conditions (improper synchronization). If the intersection is not an empty set, the system is not guaranteed to be properly synchronized.

Figure 10 gives a graphical presentation of the calculation performed by the SIGALI tool. In this figure the possible states implied by the initialization conditions are given by the hexagon, while the error states corresponding to incorrect synchronization are marked by black dots. In this case, the system cannot enter into a state where the transfers are incorrectly synchronized.

### 5.3 A VHDL Library

An alternative to the SIGNAL library is a library of timing tools written in VHDL [23]. While VHDL lacks the algebraic formalisms of SIGNAL, it offers two advantages: portability; and the capability of performing automatic synthesis. Our component model requires a language capable of expressing temporal relationships for the timing sub-element, and VHDL is capable of this.

The widespread use of VHDL for VLSI circuit simulation makes it possible to exchange VHDL code with other users. In addition, as the interface of a component is in VHDL, we hope to achieve interoperability with existing and future VHDL components written by others.

The synthesis of VHDL code into ASIC's or FPGA's should make it possible to automatically synthesize any "missing links", once a complete simulation system has been created. During implementation, any VLSI component models will be replaced by the VLSI component they represent. On the other hand, any external elements needed to complete a system, such as buffers or initialization controls, can be synthesized directly from the simulation model.

## 6 An Image-compression Application

This section gives some examples of our component model, and shows how it can be applied to ease the transition between a behavioral model and an RTL model. We will present the components as they are used in an image-compression application. We first present an overview of the principals of the image compression application, followed by component models developed from the maquette presented in section 3. The components are then integrated to form a system. The components and the application are described at two levels: a behavioral (block) level; and a circuit level.

Image compression within an image (*intra-image* compression) is often performed via a sequence of three operations: Discrete Cosine Transform (DCT); quantification; and variable length coding. We have chosen to implement the ETSI specification [10] for image compression, used to transfer images between television studios. The coder is described in 300 lines of SIGNAL for the timing information, and 2000 lines of RELACS [26] and C for the operations

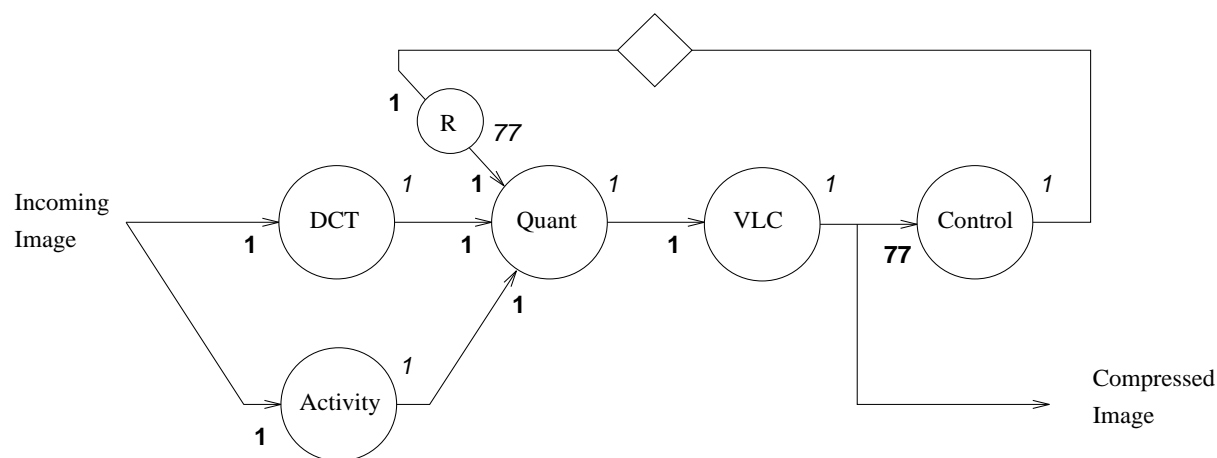


Figure 11: Image Compression Mini-coder, SDF block-level

(not including the code to do I/O on a sequence of images). In order to present a manageable example, we extract the subset of the coder which performs intra-image compression on luminance information. Because this subset is only a part of the complete ETSI coder, we refer to it as the “mini-coder”.

## 6.1 The Image-compression Mini-coder

The ETSI standard specifies a sequence of DCT, quantification, and variable length coding to perform intra-image compression, but adds two operations to control the quantification: a feedback loop from the output buffer, and an estimate of the activity in a certain block of the image. The SDF graph for this sequence is shown at a block level in Figure 11. Components which perform these operations will be used below to present the transition between behavioral and RTL system models. Timing sub-elements for these components exist at a block level and a circuit level, both of which use the same subroutines to perform the actual operations.

## 6.2 Composition of a component

We have developed components for the mini-coder using the component model presented in this article. Each component in the basic library consists of the timing sub-element written in SIGNAL, an operation written in an imperative language (i.e. a subroutine call) and interface code as needed. The components are all compiled into C code to be executed under Unix. We will present two components, a Discrete Cosine Transform and a Quantifier. Both are presented at a block level, as in Figure 14, and at a circuit level (including initialization considerations), as in Figure 15. In the following subsections, these components will be presented as they are configured for the mini-coder.

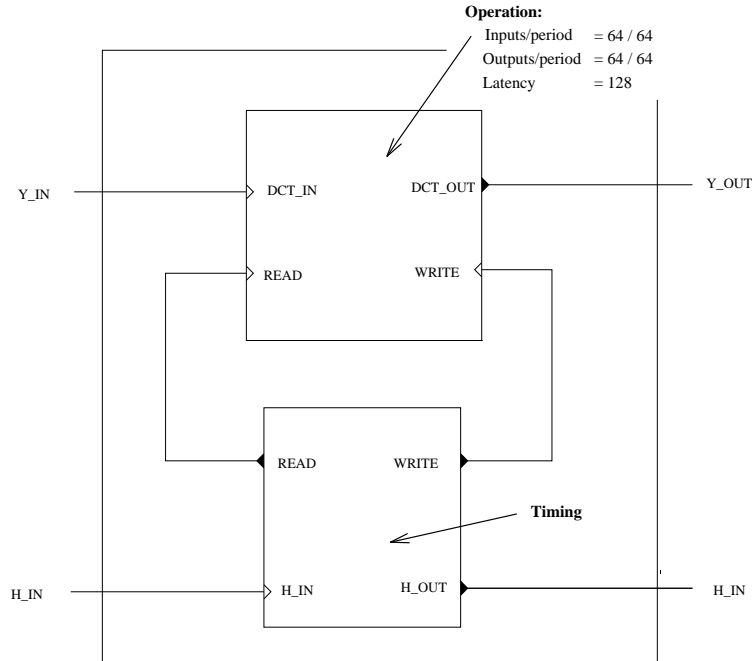


Figure 12: DCT component with timing and operation

Each component was developed using the maquette presented earlier, i.e. with an *operation* and a *timing* sub element, as well as the necessary interface between the two. We present these components as they appear under the graphical interface X SIGNAL [4], which is part of the SIGNAL real-time programming environment.

### 6.3 The DCT functional block

The Discrete Cosine Transform in our mini-coder is performed in two dimensions on 8x8 blocks of image pixels. The RELACS language was chosen to model the DCT operation for simulation. It allows us to generate code targeted at a systolic parallel processor, but which can also be executed on a workstation under Unix. In order to have a more realistic model, we implement the fixed-point operations of the inmos/SGS-THOMPSON IMS-A121 8x8 DCT chip [13]. The transform performed by this circuit uses two one-dimensional cosine transforms, the behavior of which is given by:

$$X(k) = \sqrt{2/N} c(k) \sum_{i=0}^{N-1} x(i) \cos \frac{(2i+1)k\pi}{2N} \quad k = 0, 1, \dots, N-1 \quad \text{where}$$

$$c(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } k=0 \\ 1 & \text{for } k=1, \dots, N-1 \end{cases}$$

Our model implements multiplication, saturation, and rounding operations to the same number of bits as the circuit, with the exception of the final matrix multiply, where we pass only 32 bits instead of 33. The code assumes 32 bit unsigned integers.

At a behavioral level the DCT is a subroutine, which takes an array of 64 pixels, and returns an array of 64 coefficients.

At a block level the timing sub-element of the DCT is arbitrary. For convenience, we decide that processing starts with the first block of input pixels, and the block of output coefficients is available at the output one cycle later. The subroutine which calculates the DCT yields a result without delay, and so the interface logic consists of a buffer which holds the block of transformed coefficients for one cycle.

At a circuit level, we model the interface of the IMS-A121. This processor reads one image pixel per clock, and writes one transformed coefficient per clock. Because the transform is performed as a sequence of two one-dimensional transforms, and each transform is implemented directly from the equation above with one multiply-add operation per cycle, the component has a latency of 128 cycles. The specification of the IMS-A121 requires that the circuit be idle for at least 3 blocks before the output data is considered valid. This is not specifically coded in the component, as it is assumed that the component will be part of a larger system with an initialization phase, and thus the initialization input `H_IN` will be false for sufficiently many clock cycles.

### An RTL model of the DCT

An RT level component model of the DCT was developed using the maquette and timing tools from the toolbox. An “exploded” graphical view of the resulting component is shown in Figure 12. The operation takes a block of 64 input pixels per period, and generates 64 output pixels per period. As specified in the IMS-A121 datasheet, the latency of the operation is 128 cycles at an RT level. The parameters for the timing sub-element of the DCT component at a circuit level are given in the balloon of Figure 15 as the number of input elements per period, the number of output elements per period, and the latency at the circuit level. At a block level, the latency is 1 (as with all components), and the period also occurs in one clock cycle.

Processing begins when the input `H_IN` goes true. Any input received at `DCT_IN` before `H_IN` goes true will be considered an error. Once `H_IN` goes true, an input is assumed to arrive with every clock cycle; otherwise an error results. The output `DCT_OUT` and the initialization output `H_OUT` will be valid 128 clock cycles after the input begins. There will be an output at `DCT_OUT` for every clock cycle after that, until 128 clock cycles after `H_IN` goes false. The initialization output `H_OUT` will also go false 128 clock cycles after the input `H_IN`.

## 6.4 The Quantizer

As data passes through the coder, following the cosine transform the 8x8 blocks of DCT coefficients are quantized. Quantization is based on two control parameters - an estimate

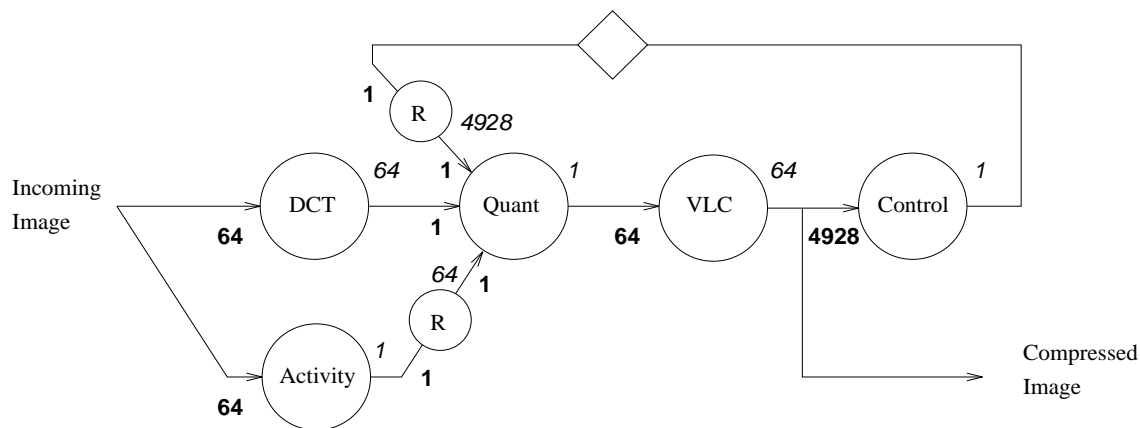


Figure 13: Image Compression Mini-coder, SDF representation

of the activity in a particular  $8 \times 8$  block, and a buffer occupancy factor. We did not base our component model on a particular circuit, but rather created a hypothetical component using the ETSI specification Annex V [10]. This hypothetical component is optimized at the circuit level to have a latency of one clock cycle (as it might be if implemented using a table lookup and some logic).

At the block level, a complete block of 64 pixels is quantized using two control values: an activity level for each block; and a buffer occupancy factor which is calculated once per stripe. Interface logic consists of a buffer which stores the result for one clock cycle, and latches which hold the activity and buffer occupancy values (and also check that the values are updated at the appropriate instants).

### An RTL model of the Quantifier

At a circuit level, we do not model an existing circuit, so we hypothesize that the quantization would be done on a word-by-word basis, to avoid unnecessary storage. This gives us a circuit model with a latency of one cycle at a word-serial level, necessitating as well a change in the functional sub-element to work on a word-by-word basis. The interface logic includes latches for the quantization control values, which also check for correct synchronization. We note that the quantizer could also have been modeled as having 64 inputs per period, and 64 outputs per period, but with a latency of one cycle, the output would be valid before all the input has been received, a logical impossibility.

The quantifier as used in the coder demands a more sophisticated synchronization than the DCT, as it requires both input data and control parameters. The input data comes directly from the DCT. The control parameters are an activity factor, calculated once per block, and a buffer occupancy factor, calculated once per stripe, i.e. a one-block-high horizontal slice of the image. In the SDF diagrams (cf. Figures 11 and 13) this “regeneration” of the control values is indicated by the circle marked **R**. Processing begins when the `H_IN` input (which is the `H_OUT` of the DCT) goes true.

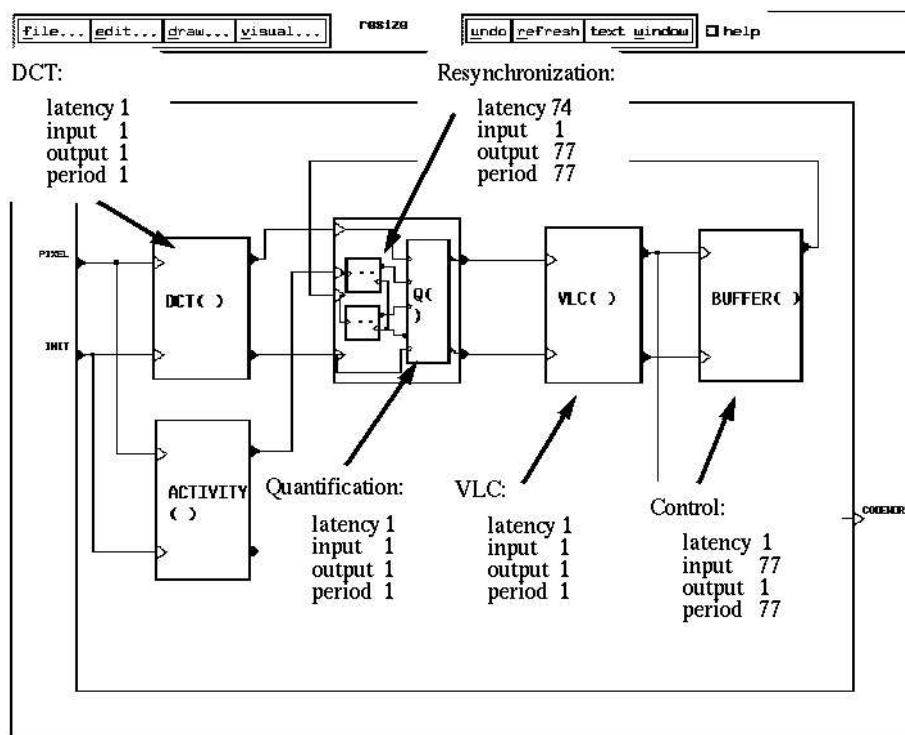


Figure 14: Image Compression Mini-coder, block level

Resynchronization blocks, using the process R, are part of the coder used for simulation, to assure that the communication is correctly synchronized during simulation. These can be seen in Figures 14 and 15 as the small blocks in front of two of the Quantifier inputs. In the following subsection we will detail the control-loop communication found in the mini-coder, and present the SIGNAL blocks used to insure that this communication is correctly synchronized.

## 6.5 The Complete Application

The components described above were developed for use in a demonstrator system for image compression. The sequence for luminance, as coded in SIGNAL under the XSIGNAL interface, is presented at a block level in Figure 14, and at a circuit level in Figure 15. We emphasize the fact that both schemas are identical at the system level; it is the component library which changes.

The integration of components into a complete system is facilitated by the synchronization support offered through our component model and tools. Nevertheless, there are system-level synchronization issues to be considered. In our example, there is a feedback loop from the output buffer to control the quantifier, which we now consider in greater detail.

## Synchronization of the Feedback Loop

The control challenge is to insure that the new value for the buffer occupancy factor is correctly synchronized with the start of a stripe, or line of blocks. A simple solution to the synchronization problem would be to latch the new value for the buffer occupancy factor when it arrives, and present it to the quantifier simultaneous with the arrival of the first block in a stripe. In order to be able to assure the correctness of the synchronization during simulation, we choose instead to specify precisely (i.e. overspecify) this property, using an R process to perform the resynchronization of the occupancy factor, and a check for proper synchronization.

The R process takes the parameters LATENCY, the number of INPUT events, the number of OUTPUT events, and the PERIOD. In this example, at a block level correct synchronization demands that the new buffer occupancy factor be delayed by 74 blocks to be synchronized with the start of a new line of blocks, and that it be sampled 77 times per line. For the SDF graph, this implies a delay indicated by a diamond in Figure 11 and a replication by the node marked “R” of 77 outputs per input. For the simulation using SIGNAL, the delay is expressed as the latency of the R process, yielding the parameters (74, 1, 77, 77). In Figure 14 the section of the coder which performs this check is shown. If the buffer occupancy factor is not correctly synchronized, an error results during simulation.

At a circuit level, transfers occur in word-serial fashion, leading to 64 cycles per block. At this level, correct synchronization demands that the new buffer occupancy factor be delayed by 4799 cycles to be synchronized with the start of a new line of blocks, and that it be sampled 64 times per block, or 4928 times per line.

Thus in this case the parameters in the SDF graph are a delay of one (corresponding to one control output value per line), and an up-sampling of 4928 outputs per input, as shown in Figure 13. For the SIGNAL circuit-level simulation, the parameters are (4799, 1, 4928, 4928), as shown in Figure 15.

We note that this approach is in fact an over-specification of the system, as it would suffice to store the buffer occupancy factor when it arrives, and update the value at the start of a new stripe. The over-specification provides an extra check of correct design.

## 7 Conclusion

This article has presented a new model for synchronous VLSI components, and an application example. A simulation library, intended to support Synchronous Data Flow design for synchronous VLSI circuits, has been introduced. The library contains components built according to our model, which specify both the operation performed by the component, and the timing diagram of each component, but without specifying the internal details of a component. The components can also be developed directly from a purely behavioral specification, allowing easy simulation with realistic computation delays.

Our approach offers advantages both in the modularity of a component and in the efficiency of system-level simulation. The components need not be described internally at a

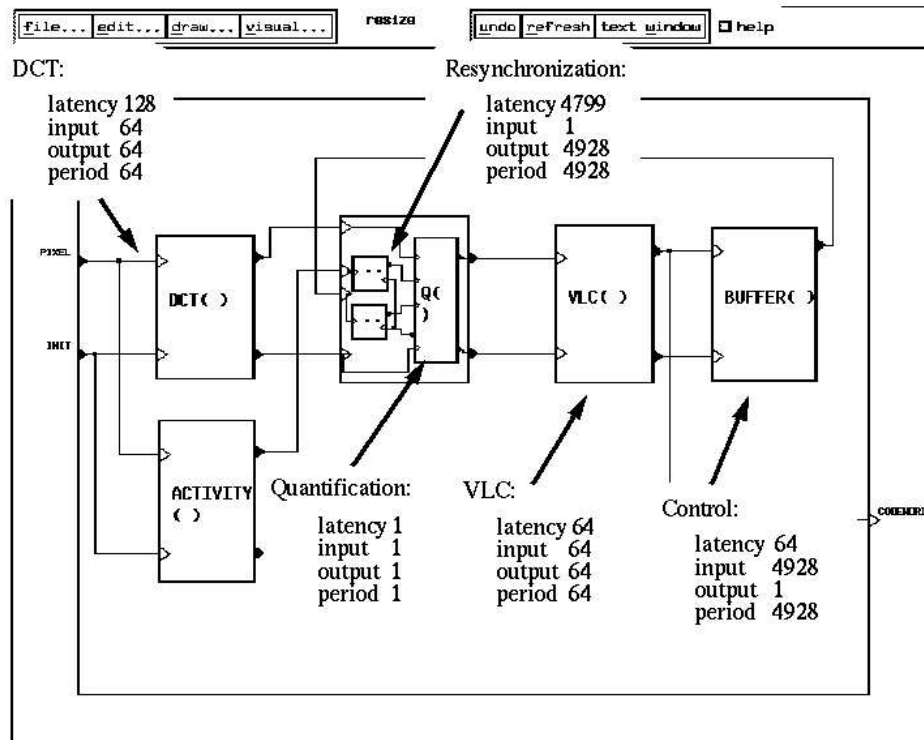


Figure 15: Image Compression Mini-coder, circuit level

gate level. And the separation of functions make it easy to change the “technology” of the simulation library. It also adapts well to the typical stages in the design process, beginning at a block-level and finishing as circuit-specific.

The component model presented proposes a transition mechanism for introducing initialization and termination considerations into a Synchronous Data Flow graph. These considerations are an important part of the detail work needed for a dedicated-component implementation, particularly when working with complex VLSI circuits. The library model assumes that each component performs a certain cyclic operation independent of the data, and that the timing of input and output events can be calculated relative to this cyclic operation using a single clock. This can be seen as a major limitation; however, it simplifies our model and makes precise synchronization possible. We also benefit from the work done on Synchronous Data Flow [21][22] and synchronous circuit design.

Our approach offers the advantages typical of multi-level simulations, both in the modularity of a component and in the efficiency of system-level simulation. The components need not be described internally at a gate level. And the separation of functions make it easy to change the “technology” of the simulation library.

## 7.1 Further Work

Current efforts are focused on the use of VHDL to describe the timing aspects of components. This article presents components in SIGNAL and C; the same tools in VHDL and C can be used in the SynOpSys environment [29].

## Acknowledgements

We would like to thank Doran Wilde for his careful reading and insightful comments.

## References

- [1] W.B. Ackerman. Data flow languages. *Computer*, 15:15–25, February 1982.
- [2] Albert Benveniste, Paul Le Guernic, Yves Sorel, and Michel Sorine. A Denotational Theory of Synchronous Reactive Systems. *Information and Computing*, 99(2):192–230, August 1992.
- [3] Michel Le Borgne. Dynamical systems over Galois fields: Applications to DES and to the SIGNAL Language. April 1993. Internal Note.
- [4] Patricia Bournai and Paul LeGuernic. *Un environnement graphique pour le langage SIGNAL*. Technical Report, IRISA, Campus de Beaulieu, Rennes, France, 1993. RR-741.

- 
- [5] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
  - [6] Joseph Buck, Soonhoi Ha, Edward Lee, and David Messerschmitt. *Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems*. Technical Report, University of California, Berkely, August 1992.
  - [7] A.L. Davis and R.M. Keller. Data flow program graphs. *Computer*, 15(15):26–47, February 1982.
  - [8] J.B. Dennis, J.B. Fossen, and J.P. Linderman. Data FLOW Schemas. *Lecture Notes in Computer Science*, 5:187–216, 1972.
  - [9] Bruno Dutertre. Spécification et preuve de systèmes dynamiques. Thèse de l'Université de Rennes 1, France, December 1992.
  - [10] ETSI. *Specification of Component TV codecs 32-45 Mbit/s*. Technical Report, European Telecommunication Standards Institute, December 1990.
  - [11] N. Ghezal, S. Matiatos, P. Piovesan, Y. Sorel, and M. Sorine. *Syndex. Un environnement de programmation pour multi-processeur de traitement du signal. Mecanismes de communication*. Research Report 1236, INRIA Rocquencourt, France, 1990.
  - [12] Paul N. Hilfinger. A high-level language and silicon compiler for digital signal processing. In *IEEE Custom Integrated Circuits Conference*, May 1985.
  - [13] inmos. IMS A121 2-D Discrete Cosine Transform Image Processor. Data Sheet, 1991.
  - [14] Alain Kerihuel, Roderick McConnell, and Sanjay Rajopadhye. Des graphes de flots de données synchrones pour le vlsi. June 1994. (à paraître).
  - [15] Alain Kerihuel, Roderick McConnell, and Sanjay Rajopadhye. VSDF: Synchronous Data Flow for VLSI. In *Proceedings of the 37th Midwest Symposium on Circuits and Systems*, Lafayette, Louisiana, August 1994. (to appear).
  - [16] Kernighan, W. Brian, and D. M. Ritchie. *The C programming language*. Prentice-Hall, 1978.
  - [17] Christophe Lavarenne and Yves Sorel. Spécification, optimisation de performance et génération d'exécutif pour application temps-réel embarquée multi-processeur avec syndex. In *CNES International Symposium*, Les Saintes-Maries-de-la-Mer, November 1992.
  - [18] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79, September 1991.

- [19] Edward A. Lee. Consistency in Dataflow Graphs. In *IEEE Transactions on Parallel and Distributed Systems*, pages 355–369, IEEE Computer Society, April 1991.
- [20] Edward A. Lee. Multidimensional Streams Rooted in DataFlow. In *IFIP Working Conference on Architectures and Compilation Techniques of Fine and Medium Grain Parallelism*, North-Holland, January 1993. Orlando, Florida.
- [21] Edward A. Lee and David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, C-36(1), January 1987.
- [22] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9), September 1987.
- [23] *IEEE Standard VHDL Language Reference Manual*. Institute of Electrical and Electronics Engineers, New York, 1987. IEEE Std 1076-1987.
- [24] E. Martin, O. Sentieys, H. Dubois, and J. Philippe. GAUT: an architectural tool for dedicated signal processors. In *Euro-DAC '93*, pages 14–19, IEEE, Hamburg, September 1993.
- [25] Roderick McConnell, Alain Kerihuel, and Frédéric Raimbault. Tools for Correct DSP Synchronization. In *Proceedings of the 1993 IEEE Workshop on VLSI Signal Processing*, pages 206–214, Koningshof, Velhoven, The Netherlands, October 1993.
- [26] F. Raimbault and D. Lavenier. ReLaCS for Systolic Programming. In *International Conference on Application Specific Array Processors*, Venice, October 1993.
- [27] Jan Rosseel, Guy Lauwers, and Francky Catthoor. *Array clustering in the context of the CATHEDRAL-IV environment*. Technical Report PPR 3, NANA-I, March 1992.
- [28] Charles Seitz. *System Timing*, chapter 7, pages 218–254. Addison-Wesley, 1980. In Mead and Conway, *Introduction to VLSI Systems*.
- [29] *VHDL System Manual V3.0*. Synopsys, Inc., 1992.
- [30] J. van Meerbergen, P. Lippens, B. McSweeney, W. Verhaegh, and A. van der Werf. *PHIDEO: High-Level Synthesis for High Throughput Consumer Applications*. Technical Report, Philips Research Laboratories Eindhoven, November 1992. Submitted for the Journal of VLSI Signal Processing, special issue on DSP synthesis.
- [31] J. van Meerbergen, P. Lippens, B. McSweeney, W. Verhaegh, A. van der Werf, and A. van Zanten. Architectural strategies for high-throughput applications. *Journal of VLSI Signal Processing*, 5(2/3):201, April 1993.



Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,  
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY  
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex  
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1  
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex  
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

---

Éditeur

INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)

ISSN 0249-6399