Cours/Lecture Series

1986-1987 ACADEMIC TRAINING PROGRAMME

LECTURER : P. DE WILDE / Technological University, Delft

TITLE : Computer-aided engineering in electronics

DATES : 26, 27, 28 November, 1986

TIME : 11h00 to 12h00

PLACE : Auditorium

ABSTRACT

- *Integrated electronics : technological families, emphasis on CMOS and NMOS.*
- *Computer-aided circuit design : the design trajectory.*
- *Synthesis : Si-compilation.*
- *Analysis : modelling, verification at the different levels.*
- *Modern issues : intelligent data management.*

• LECTURE NOTES

SIS/H/3416

Hierarchical Design of Processor Arrays Applied to a New Pipelined Matrix Solver†.

P. Dewilde, J. Annevelink, E. Deprettere
and K. Jainandunsing

Department of Electrical Engineering
Delft University of Technology
Delft, the Netherlands

ABSTRACT

The computer-aided design of dedicated pipelined processors for numerical applications and signal processing requires design tools that support system refinement, partitioning strategies and the transformation of behavioral descriptions into structure. In this paper we present an hierarchical design methodology, and apply it to the design of a pipelined matrix solver.

1. INTRODUCTION

Digital signal processing often requires very fast handling of data in a small physical area and with low power consumption. A good example is that of bit-serial digital filters which are capable of achieving high throughputs with a very limited area, taking into account numerical considerations like stability and accuracy. In such very dedicated applications, the algorithms used can be optimally adapted to the requirement of hardware minimization within the speed constraints. A prototype example of this technique can be found in [1]. In the present paper we wish to take a more general point of view where we develop the design path from a numerical algorithm down to hardware with the expressed goal of realizing a structure that exhibits a number of desirable qualities: regularity, simplicity of control and high performance. Our interest will go to the definition of a design methodology which can handle a broad class of algorithms, such as a matrix solver or a system to compute eigenvalues. The methodology will be set up in such a way that at the limit any deterministic system can be so designed. However, we shall concentrate on the problems associated with mapping algorithms on regular, dedicated hardware: hierarchical refinement, partitioning and the generation of control structures. The emphasis will be on placing all the components of the design system in one consistent framework. As we shall see, such a consistent framework is obtained by combining and modifying classical concepts so as to fit the general design situation. Foremost is the notion of *signal flow graph* which is

† This research has been partly supported by the commission of the EC under ESPRIT contract 991

classical in signal processing, but needs modification as will be explained further. The model that we shall use to describe system structure and behaviour is based on the notions of *functional calculus* and *applicative state transitions*, which are also (to a good observer) classical to signal processing and have been rediscovered (of course with a different terminology) in computer science [2,3]. In the meantime both notions have obtained considerable attention especially in the computer science literature [4,5,6]. On the other hand, the development of description languages for signal processing has also been considered - a prime example of which can be found in [7]. Needless to say, we have used freely some ideas already present in the literature, while devoting most of our attention to the development of a consistent (and novel) design system - the design system HIFI [8].

In the course of refining a design several actions have to be taken, which the design system should support:

1. Making the type of the data that is transferred between components more refined (e.g. from "matrix" to "sequence of columns", to "sequence of floating point numbers", to "sequence of bits").
2. Decomposing components into regular or irregular structures of smaller components (which then in turn can be refined).
3. Clustering components together and mapping them onto a new set of components whose behavioral descriptions and schedules realize the original components sequentially (partitioning).

It is seen that any major design stage consists of a refinement phase (in which the components of an algorithm are made explicit) and a partitioning phase (in which the algorithmic components are mapped on a specific architecture). When a functional style is used, even algorithmic descriptions will take the form of a structural description with attached schedule. Hence it will be possible to use a uniform method of description throughout.

The HIFI system is based on an analysis of the various objects which the system designer (through his design system) has to handle, together with their relations and properties (section 2). The link between the behavioral description of a system that is supposed to realize an algorithm, and its structure (which we shall represent by an extended signal flow graph, in short an SFG) goes via an intermediate representation in which the content of the algorithm is made completely explicit - a *dependence graph (DG)* (it will turn out that the DG is itself an SFG). The DG representation is needed to allow for partitioning and scheduling in which tasks are assigned to devices, and schedules are generated. This method was originally presented in [9] and is based on the work of [10]. In recent years there has been a lot of interest in the mathematics for partitioning of regular DG's, see [11,12,13,14]. All these partitioning techniques will fit nicely in the HIFI system as shall be demonstrated by example in section 3. We shall use an attractive LISP dialect (Scheme [15]) to formulate behaviours. An alternative approach would be to start from an algorithmic description in an imperative language, say FORTRAN, and generate from it a so called "single instruction set" description [11]. This approach is conceptually equivalent to the one presented here but not as flexible (LISP allows e.g. for the expression of constraints). It is also prone to error due to hidden states (or equivalently side effects). The advantages of a functional style have been well documented in the literature, see

[2].

2. SYSTEM DESIGN BY REFINEMENT

One may define systems' design as the successive refinement from behavioral descriptions to structure, a process that finally ends in the complete definition of the hardware. This is especially true for the design of dedicated concurrent systems, like signal processors or matrix solvers as one has in speech coding, image coding or simulation [16,17]. For these applications, it turns out that the complexity of the overall solution is often strongly dependent on the organization of the dataflow and not only on the computational complexity. Examples can be found in [18,19,20].

Conceptually, the design of a system can be split in two phases:

1. the specification of the algorithm and
2. the mapping of the algorithm onto a particular hardware architecture.

The gist of the HIFI design methodology is the translation from a *semantic level* of direct I/O functional specifications to a *structural level* where states become explicit and the algorithm gets decomposed in structural parts (construction and composition). This decomposition can then be hierarchically refined without modification of the basic ideas, all the way to a hardware description. In this fashion a consistent *orthogonalization* of semantics (behavior) and structure is obtained.

A "deterministic system" is by definition a system for which there exists a (partial) map between inputs viewed globally (i.e. over all time and space) and outputs viewed globally. Hence it is an object that consists of a set of inputs, a set of outputs and a global I/O map. Inputs and outputs are characterized by a *type* which describes their structure. The I/O map, which can be viewed as "formal semantics" [21], will normally be given in the form of a *behavioral description* - in our system in LISP. It will be the task of the designer to distinguish subfunctions and extract causal relations from the behavioral description and to map these onto available devices. We shall represent a deterministic system with its attributes (input, output, constraints and behaviour) by a *node*. In the first phase of a design procedure, the refinement phase, this node will gradually be decomposed into a dependence graph of nodes containing subfunctions and edges which indicate the dependencies between the tasks and the propagation of data. Hence, the edges can also be interpreted as carrying relevant data to the nodes. In a second phase, the partioning phase, primitive functions in the dependence graph are clustered together into larger nodes, intermediate data is assigned to registers or storage, and schedules are derived. Partitioning methods can be either for general situations [10,9], or tailored to regular algorithms[12,11,22,23,13,24].

A schematic representation of the overall design procedure is as shown in Figure 1. In the forthcoming formalization, we enforce the following requirements:

1. The description must be uniform across design levels.
2. Every description must be self-contained (and hence can be simulated). Control and states at the given level must be explicit. Everything else is functional (i.e. described purely by mathematical functions) and attached to the nodes as "semantics".

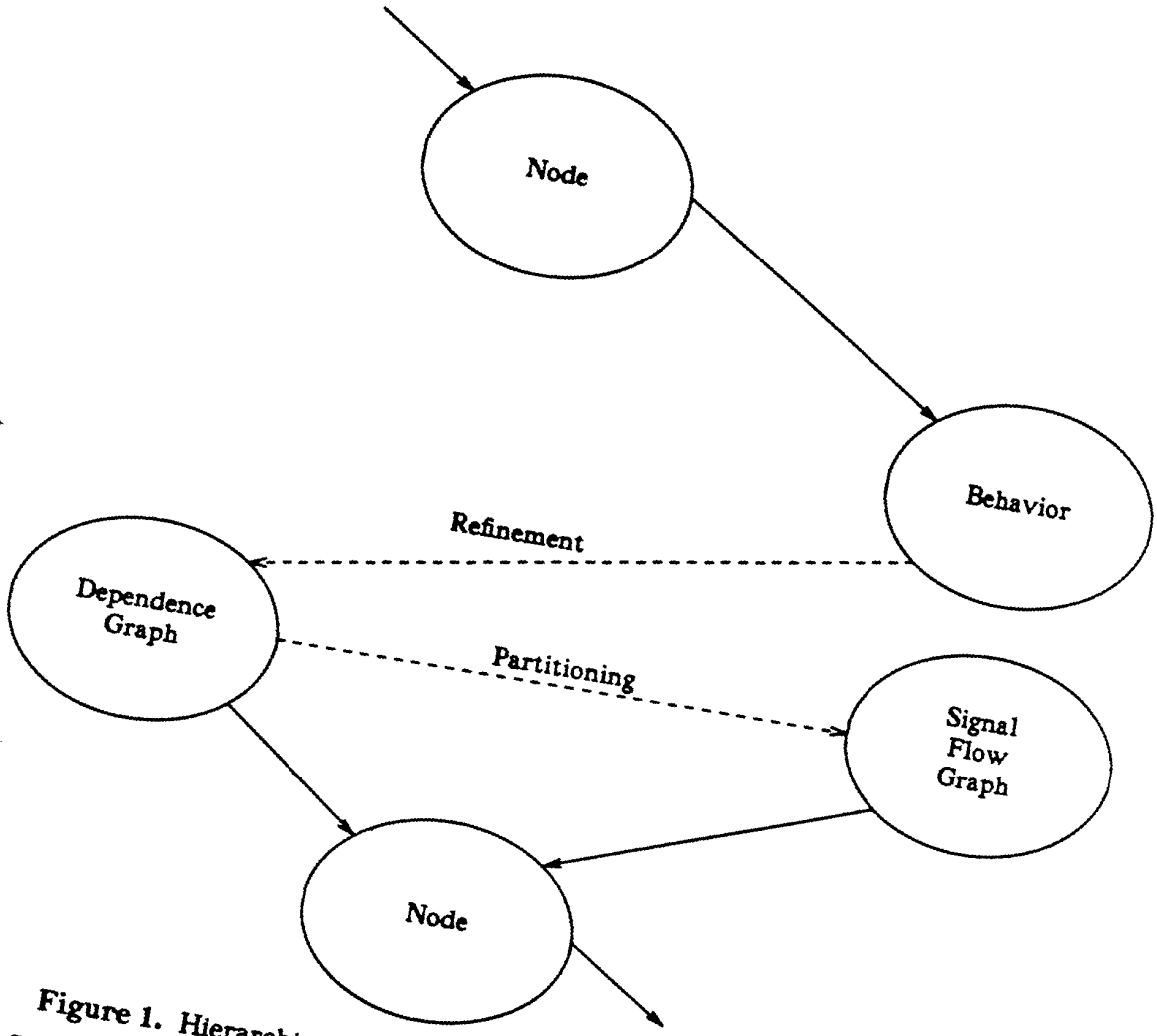


Figure 1. Hierarchical Design Procedure: from semantics to structure

3. The model must be able to cover all deterministic systems.

A computational model based on dependence graphs will not be general enough to cover all deterministic systems (there are many deterministic algorithms that cannot be represented by a dependence graph - especially those that require dynamic (data-dependent) branching, that leads to dynamic routing between devices. The possibility of representing an algorithm with a dependence graph is roughly equivalent to having a structured program, see [25]. The more general model that is still "functional" and appears to be adequate for all deterministic systems is based on the "Applicative State Transitions (AST)" of [2]. Any node in the system will be assumed to have such a description. The design is terminated when a description is reached in which the nodes at the leaves of the refinement tree are directly implementable by (VLSI) hardware.

We proceed with the introduction of the basic components of the design system that realize the previous objectives.

2.1 Behavioral Description in Nodes

Underlying any (structured) system description there is an Applicative State Transition system [2]. Although this assertion cannot be proven except in a carefully defined mathematical environment, we take it as a working hypothesis, motivated by the literature mentioned. (A proof would amount to reversing the constructive arguments that lead to set theory). Therefore, a node that was characterized before by a behavioral description, e.g. a sequence of LISP expressions, has an underlying AST description which decomposes it into (dynamic) sequences of functions. Hence to a node we can attach:

1. $F = \{f_\alpha\}$ - a set of functions which the node is able to execute (e.g. any internal state is captured in a functional way). Each f_α is a partial function: it maps some inputs to some outputs.
2. I a collection of input ports characterized by a (fixed) type.
3. O a collection of output ports also characterized by a (fixed) type.

In any of its histories, the node will execute a sequence of functions:

$$f_{\alpha_0}, f_{\alpha_1}, f_{\alpha_2}, \cdots$$

with each $f_{\alpha_i} \in F$. When f_{α_i} is executed, we say that we stand at "event i".

Let $V_I(i)$ be the values (ev. empty) of the input ports at event i, and let $V_O(i)$ be the values (ev. empty) that the output ports will obtain at event i. Characteristic for the Applicative State Transition (AST) mechanism is the definition of the maps f_α:

$$f_\alpha : V_I \rightarrow V_O \times F : \text{input} \rightarrow (\text{output}, \text{successor—function})$$

2.2 The Structure of Concurrent Systems

At any given level of the design hierarchy, the system structure is represented by a Signal Flow Graph (SFG) which makes functional relations and state that are relevant at this level explicit. The communication mechanisms used in our SFG's must be consistent with our previous AST description, and are as follows:

1. Self-timing by a single token pass discipline. For each node in the SFG, an actual partial function f_α may fire when the two following conditions are satisfied: (a) tokens are present on relevant inputs; (b) relevant outputs are free from tokens - see Figure 2.

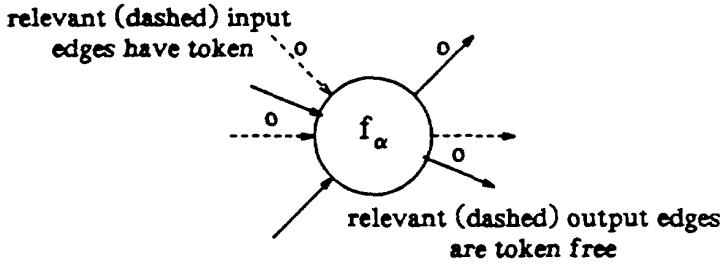


Figure 2. Node Firing Discipline

2. The state that is explicit at this level of the design is denoted by a simplified type of node that is marked as "*delay*". Because its function is exclusively to provide for memory, its AST description will consist of as many elementary functions as there is distinct data to be stored: for each piece of data one function f_{data}, with the function map:

$$f_{data} : I \rightarrow O \times F : in \rightarrow (data, f_{in})$$

While this description is mathematically sound, it is of course too cumbersome for design purposes. Hence, we single it out as one (among others to come) of the basic building blocks, for which we have a special graphical representation. It is interesting to note that the path from simple to associative memories appears to be just an extension of the data-functions by which the memory is defined.

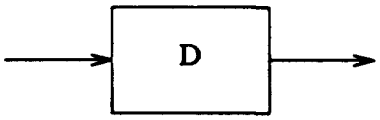


Figure 3. State Representation

This SFG model requires timing verification because nodes are in no way synchronous, and mismatches between inputs and outputs are possible. We consider this a desired property, because it catches the relevant design problem at the present level of abstraction. There are two ways in which an SFG can be incorrect: (1) a misfit between the active function at event i and the supply of tokens at the input and (2) there might be delay-free closed loops (deadlocks). We shall say that an SFG is *correct* if none of these two faults occur. The following property then holds:

If all the nodes in a SFG are AST systems and if the SFG is correct, then it is itself an AST system.

An informal proof of the property is as follows: Consider first initial functions in all the nodes. Count only active input and output edges. Since the resulting graph is acyclic, there will be maximal elements. Place tokens on the active inputs of the SFG. If none of the maximal elements is able to fire, then there is a misfit between the supply of tokens to the SFG and its nodes, violating one of the conditions. Hence, at least one maximal node, and possibly more, will be able to fire. They will propagate input tokens to the next level. The nodes that just have fired will have token free inputs. Place them in a set of "free nodes". The remainder graph will still be acyclic, and will again contain maximal nodes. Again because of the absence of misfits, one of the two following situations will occur: either at least one node will be able to fire, or all tokens will be on outputs of the SFG. In the former case we may proceed recursively, adding nodes that have fired to the set of free nodes. In the latter case all remaining nodes and edges are token free are token free and can also be added to the set of free nodes. The nodes that have fired have replaced their function by a new function. The first overall f_{α_1} has been reached, and the situation is now equivalent to the original. Proceeding recursively, we obtain f_{α_1}, f_{α_2}, f_{α_3}, ... and the SFG is itself an AST system. Because of it,the method proposed here is internally consistent.

A further classification of AST systems is possible. In the generality presented above, no fixed schedule of operations need be possible. It is easy to conceive AST systems in which the order of the operations is so much dependent on the data that no reasonable operational description can be given. Such systems can only be verified through exhaustive tracing. The other extreme - the one we consider further - are AST systems for which an a priori scheme of operations exist in the form of a dependence graph. Such systems we shall call "static" (there is no dynamic routing of data). They can be verified and synthesized a priori, and can hence be the subject of a structured design method.

To illustrate the theory so far, we terminate this section with a simple example showing the concepts used. In the next section a more sophisticated example will be presented.

Example 1: compute square root of an integer.

We start out from a description of the algorithm in (self-explanatory) LISP † (the behavioral description) and proceed through refinement and partitioning steps until a target architectural description has been reached. Let's use the following simple minded algorithm:

```
(define (hf-sqrt in n)
  (sqrt-iter 0 in (expt 2 (quotient n 2))))

(define (sqrt-iter est in incr)
  (let ((sq-est (* est est)))
       (cond ((or (= sq-est in) (= incr 0)) est)
             (else (if (< sq-est in)
                       (sqrt-iter (+ est incr) in (quotient incr 2))
                       (sqrt-iter (- est incr) in (quotient incr 2)))))))
```

The algorithm proceeds in a fixed number of steps (depending on the precision n of the integers used) by adding or subtracting decreasing powers of two (incr) to or from a running estimate (est). Given the algorithm (it could also, but maybe not equally nicely, have been specified in another language), we proceed by moving into a HIFI environment (typically on a workstation). At the top level we use the graphics editor to define a class of nodes (keyword: node-class) parameterized by n, a scalar, which will represent systems realizing the desired algorithm (Figure 4). The editor allows the designer to define the inputs, outputs and other attributes of the top-level node. The editor is invoked by evaluating the expression:

```
(make :node-class hf-sqrt)
```

A textual representation of the object thus created is given below:

```
(node-class (hf-sqrt (:scalar n))
  (inputs (:scalar in))
  (outputs (:scalar root))
  (behavior (sqrt in)))
```

The designer can associate a behavior with a HIFI node. It is defined in a similar

† In this and other examples, we will use a simple and elegant dialect of LISP, SCHEME [15].

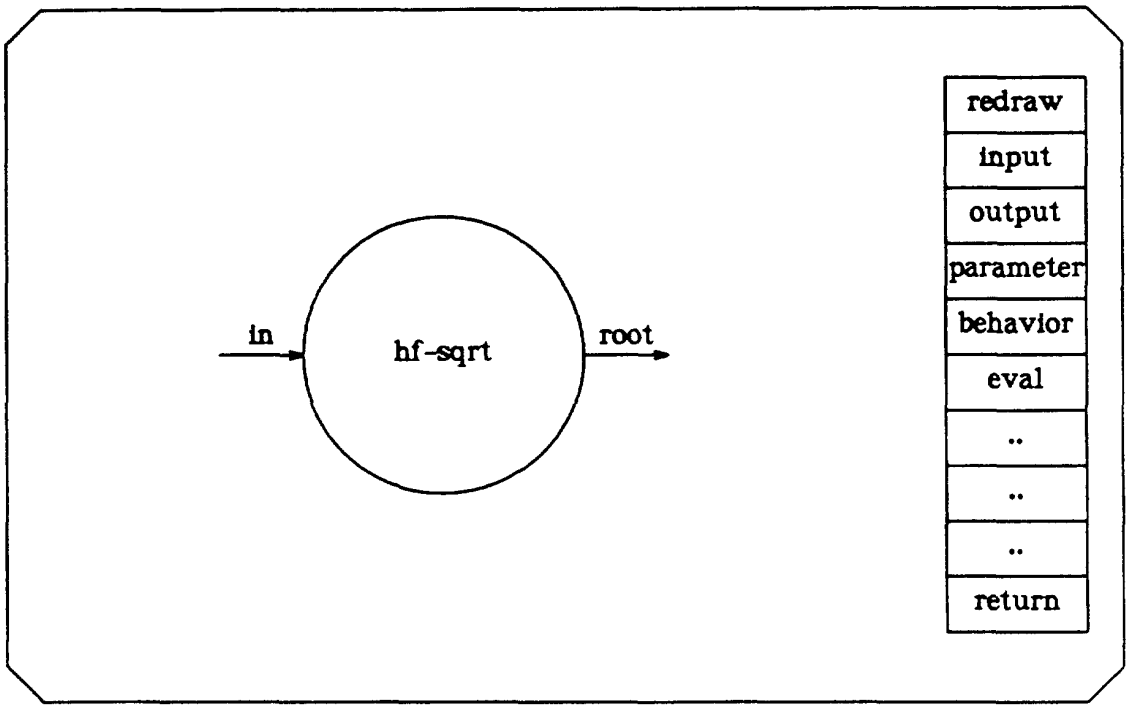


Figure 4. Initial Node Definition

way to the body of a LISP procedure, i.e. as a sequence of LISP expressions. The values returned as a result of evaluating the last expression are assigned to the outputs of the node. Given a behavior it is possible to simulate a node by evaluating its behavior. A node will actually be simulated when the designer selects the item *eval* from the editor menu.

In the case just shown, the behavior is implemented by calling the standard LISP function sqrt. The LISP functions are normally defined in the system global environment, whereas a node-class is defined in a node environment. Node environments can be created by the designer and serve as a flexible library mechanism. By doing so, there can also be no confusion of names, e.g. hf-sqrt in the example above is bound to a LISP procedure object in the system global environment, and to a node-class object in the current node environment. Depending on the environment in which the symbol hf-sqrt is evaluated the designer will get either the procedure, or the node-class.

To realize the algorithm described above, a better idea is to specify the behavior using the procedure sqrt-iter:

```
(behavior (sqrt-iter 0 in (expt 2 (quotient n 2))))
```

Since the procedure hf-sqrt, as defined above, is *tail-recursive*† one will be able to

† A tail recursive procedure is a procedure that applies itself just before returning to its calling environment. (see e.g. [15]).

implement it iteratively. This can be done with a standard HIFI procedure, called "regular refinement", which will implement the node via repeated application of a (to be defined) subnode implementing the iteration. In the HIFI system regular refinements are specified graphically. The designer first creates a, further unspecified, regular refinement of a node, using the procedure *make-regular-refinement-of*. Next this refinement can be edited. Initially the editor will display an elliptical boundary, denoting the node being refined. At the boundary we find the inputs and outputs of this node. In the center of the refinement the designer sees two small circles. These circles are the iteration nodes. We need two instances, so as to be able to specify the mutual interconnections. The designer can change the iteration node by selecting the item *inode* from the menu appearing at the right side of the display. When doing so a new menu will appear, that contains the nodes, i.e. their names, as known to the HIFI system. The designer can select any of these nodes as the iteration node. After selecting the appropriate iteration node the designer may then proceed by interconnecting the inputs and outputs of the nodes that are instances of the iteration node with each other and with the inputs and outputs of the node being refined.

We obtain (interactively) Figure 5, which describes a (parametrized) class of SFG's suitable to implement the algorithm. In Figure 5 we see small circles attached to some inputs and outputs. These circles denote sources and sinks respectively. The former just generate initial data, while the latter consume tokens. The data generated by a source can be specified by the designer as a single LISP expression.

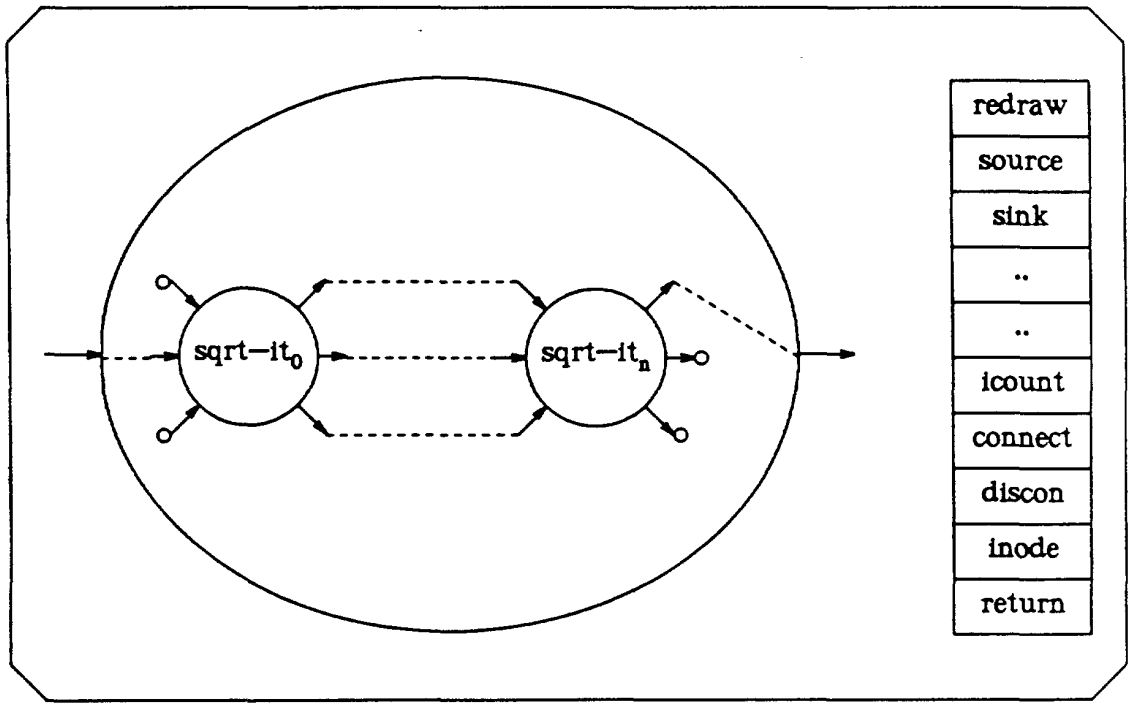


Figure 5. Regular refinement of *hf-sqrt*

The iteration node sqrt-it is defined by the node-class:

```
(node-class (sqrt-it)
  (inputs (:scalar est in incr))
```

```
(outputs (:scalar est+ in incr+))
(behavior (let ((sq-est (* est est)))
     (cond ((= sq-est in) (list est in incr))
           (else
            (if (< sq-est in)
                (list (add est incr) in (quotient incr 2))
                (list (sub est incr) in (quotient incr 2)))))))
```

This step is the "refinement procedure" for this case. It can be seen that the (parametrized) SFG in Figure 5 is actually a (primitive) dependence graph.

The next design step will be "partitioning". Suppose that we wish to realize our algorithm in one processor. Then all the present nodes have to be mapped on one with an appropriate schedule (i.e. AST description). Figure 6 shows the result.

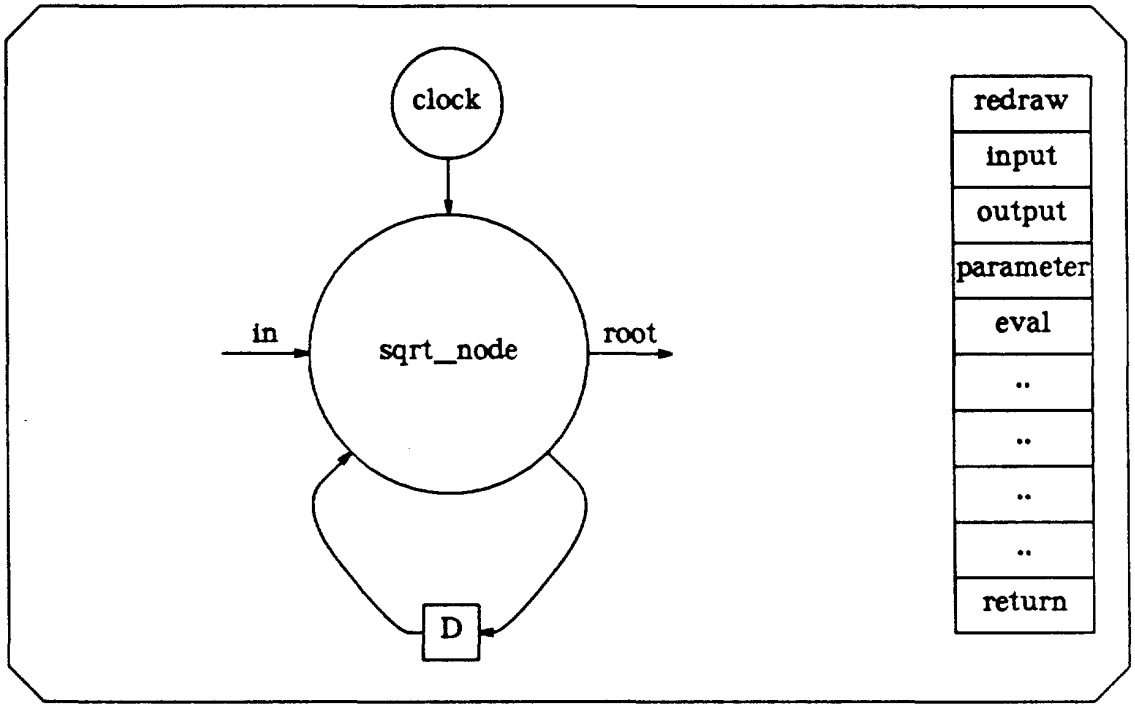


Figure 6. Partitioning of the regular refinement onto one processor

Note that we make use of a standard HIFI node, clock. A clock is defined as an input-less node that sends out originally *initial* and next starts counting modulo n. The final token in each cycle of n tokens is marked *final*. The value of n is a parameter of the clock node. Each of the nodes, including the standard ones, will have an appropriate HIFI description of its connections and behaviour, the description for the central node "sqrt_node" being:

```
(node-class (sqrt-node)
  (inputs (:scalar in) (:record state) (:clock clock))
  (outputs (:scalar root) (:record state))
  (ast
   (function (init-sq n)
             (inputs in)
             (outputs state)
             (next-function sq-it)
             (behavior
              (make :record 'est 0 'in in 'incr (expt 2 (/ n 2)))))
```

```
(function (final-sq n)
          (inputs state)
          (outputs root)
          (next-function init-sq)
          (behavior
           (member-of state est)))
(function (sq-it n)
          (inputs (member-of state est) (member-of state in)
                  (member-of state incr))
          (outputs (member-of state est) (member-of state in)
                   (member-of state incr))
          (next-function (if (= clock 'final)
                             final-sq
                             sqrt-it))
          (node-class sq-it))
(initial-function init-sq)))
```

Remark that all the node descriptions are independent of each other and self-contained so that further refinements can take place autonomously. Note also that although inputs and outputs can have similar names, this does not imply any relation whatsoever.

The delay edge appearing in Figure 6 is defined by an expression connecting the state output and input of the node sqrt-node:

```
(connect (formal-arg 'state (outputs-of sqrt-node))
         (formal-arg 'state (inputs-of sqrt-node))
         (make :record 'est () 'in () 'incr ())
         (delay 1))
```

The consistency of the nodes and refinements is maintained by means of a constraint propagation mechanism. This will be explained in more detail in the second example.

3. SOLVING SETS OF EQUATIONS IN A PIPELINED FASHION

Classical algorithms for solving systems of linear equations of the type $A\,x = b$ compute the factorization of the matrix A to produce an upper triangular system which is then solved by a procedure called "backsubstitution". The resulting dataflow is very unfavourable for parallel processing because the backsubstitution step needs the data outputted by the factorization step in reverse order. To overcome this problem an algorithm which solves the system in one pass, thereby avoiding the backsubstitution step, was first proposed in [26] and will be presented here. The algorithm does not require any intermediate accumulation of data, and is ideally suited for implementation on a dedicated array of processors. We also show how the algorithm is mapped to the VLSI array, after further partitioning.

Thus, given is a system of linear equations $A\,x = b$ where the matrix A is $n \times n$ and b a vector of dimension n. The traditional method of solving the system is by factoring A as $A = QR$ where Q is a transformation matrix which we choose to be orthogonal for numerical accuracy and R is uppertriangular. If b is likewise transformed to $\beta = Q^t b$, then the system of equations is transformed to $Rx = \beta$ and x is found by backsubstitution on β. The latter operation starts with the last row in R, while the factorization produces the first row first. A conceptual architecture (also represented by a SFG) representing these operations is shown in Figure 7.

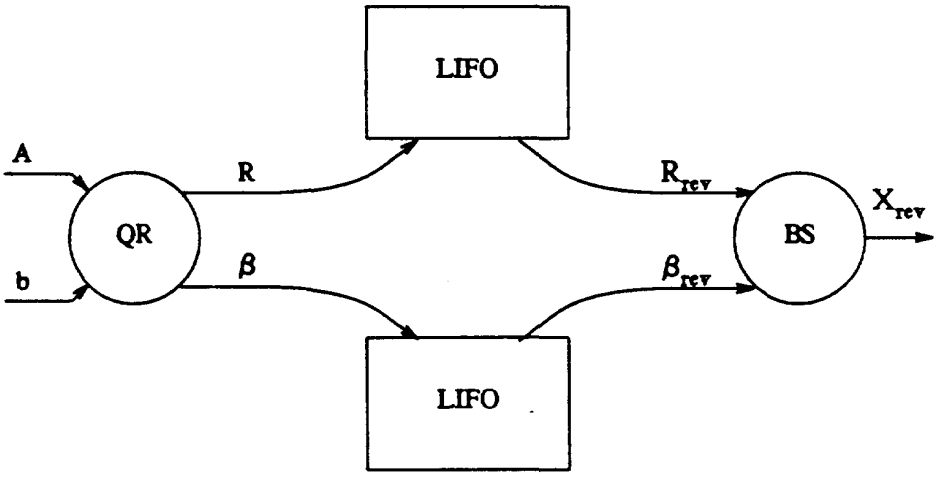


Figure 7. Architecture of the Classical Matrix Solver

By a clever arrangement of the data it is, however, possible to restrict the operations to factorization only. Inspired by the work of Faddeeva [27]. who presented a Gaussian algorithm which incorporated the backsubstitution, and following [26], we factorize the matrix:

$$A' = \begin{bmatrix} A^t & I \\ -b^t & 0 \end{bmatrix}$$

With appropriate partitioning of the matrices we obtain:

$$\begin{bmatrix} U_{11} & u_{12} \\ u^t_{21} & u_{22} \end{bmatrix} \cdot \begin{bmatrix} A^t & I & 0 \\ -b^t & 0 & 1 \end{bmatrix} = \begin{bmatrix} R' & U_{11} & u_{12} \\ 0 & x^t u_{22} & u_{22} \end{bmatrix}$$

where $\begin{bmatrix} U_{11} & u_{12} \\ u^t_{21} & u_{22} \end{bmatrix}$ is an orthogonal matrix and R' is an upper triangular matrix.

The operations performed during the factorization follow the classical Householder algorithm for which we refer to [28].

3.1 The orthogonal Faddeeva Algorithm

The algorithm, which we will call *orthogonal Faddeeva*, itself consists of two parts:

1. Create a (n+1)×(2n+1) matrix A′, from A and b, the arguments to the Faddeeva procedure. The matrix A′ is constructed by appending the transpose of b as the $n+1^{th}$ row to A, and then appending the unity matrix I of dimension n + 1 to it.

2. Application of a recursive procedure qr-iter.

The procedure *qr-iter* reflects the first column of its first argument, a matrix, computing a reflection vector q. It then reflects the columns of the argument matrix. Then it recursively calls itself on the transformed matrix, which is first reduced, i.e. the first row and column are stripped off. The recursion stops when the matrix has only one row left. In that case the procedure returns the (transformed) elements of the last row.

The more interesting part is clearly the iteration procedure, which is easily specified in the form of a LISP procedure.

```
(define (faddeeva a b)
  (define (qr-iter a)
          (let ((q (vector (car a))))
               (let ((a+ (mapcar (lambda (col) (reflect col q))
                                 (cdr a))))
                    (if (single-row? a)          ; last iteration
                        (mapcar car a+)
                        (qr-iter (mapcar cdr a+))))))
  (qr-iter (append (mapcar append a (transpose b))
                   (unity-matrix (1+ (dimension a))))))
```

Looking at its definition we notice that, although the procedure is specified recursively, it can be executed iteratively. This is because it is a so-called *tail-recursive* procedure [15]. In the above procedure we assume that a matrix is represented as a list of (column) vectors, each of which is a list of elements. As a result we can easily manipulate vectors and matrices, using the standard LISP list manipulation procedures *car* and *cdr*, where *car* returns the first element of a list and *cdr* a list containing all elements, except the first. Another basic LISP procedure is *mapcar*, which applies a procedure, its first argument, repeatedly to the elements of a list, its second argument, collecting the results in another list,

which is returned as the result of the procedure. Within the Faddeeva procedure we also make use of the procedures vector and reflect. Given the vector v, *vector* computes a vector q which represents the elementary reflection matrix,

$$Q = I - \frac{2qq^t}{\|q\|^2}$$

which transforms v into a vector in the direction of the first unit vector. *Reflect*, when applied to an arbitrary vector x, performs the actual reflection. By representing Q as a vector, the multiplication can be done efficiently. The LISP code for these procedures is given below:

```
(define (reflect x w)
  ; compute the Householder reflection of x with respect to w.
  ; input:       x       - input vector
  ;              w       - car:  norm-square of the reflection vector
  ;                      - cdr:  reflection vector
  ; output:              - the reflected vector
  (define normw (car w))
  (set! w (cdr w))
  (define x.w (* 2 (/ (innerproduct x w) normw)))
  (mapcar (lambda (xi wi)  (- xi (* x.w wi))) x w))

(define (vector a)
  ; compute the Householder reflection parameters
  ; input:
  ;     a       - input vector
  ; output:
  ;             - list car : norm-square of reflection vector
  ;                   cdr : reflection vector
  (let ((wnorm2 (innerproduct (cdr a) (cdr a)))
        (a1 (car a)))
       (let ((w (cons (+ a1 (* (if (< 0 a1) 1 -1)
                               (sqrt (+ wnorm2 (* a1 a1)))))
                      (cdr a))))
            (cons (+ wnorm2 (* (car w) (car w)))
                  w))))
```

The other procedures are simple: *transpose* transposes a vector, returning a matrix that has one row and as many columns as the vector had elements, *unity-matrix* returns the I matrix of dimension n, where n is the value given by its argument, *dimension* returns the number of columns of a matrix, the predicate *single-row?* returns true when the matrix that is its argument has a single row, i.e. all columns have one element.

Given the LISP procedures, we can start to specify the HIFI nodes and refinements. The first step is to define a node representing the Faddeeva algorithm at its most abstract level, i.e as a function that maps a matrix A and a vector b to a vector x, with:

$$A\,x = b$$

This is done interactively (cf. Figure 8) whereby the attributes of the Faddeeva node are gradually specified when they become known.

```
(node-class (Faddeeva)
  (inputs (:matrix A) (:vector b))
  (outputs (:vector x)))
```

The above definition is simple, although it already has some specific information associated with it, i.e that the inputs and outputs of the Faddeeva node are of type :matrix and :vector.

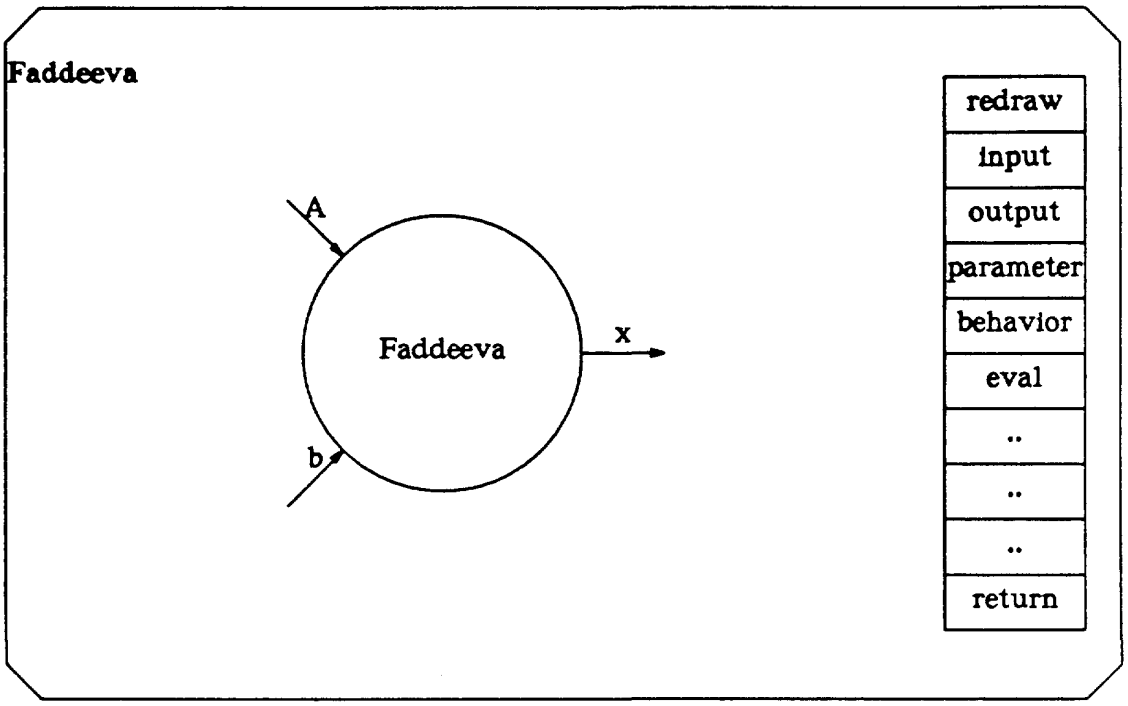


Figure 8. Faddeeva: top level node

It is possible to add more specific information to the definition of a node by adding assertions, which specify constraints on the value or other properties of inputs or outputs. When defining types, we can also specify a set of properties; e.g. the type :matrix has properties *rows* and *cols*. Thus it is possible to specify e.g. the dimension of the matrix A.

We can also specify parameters for nodes. The difference between a parameter and an input is that the value of a parameter is ultimately specified by the designer, whereas input values are unknown to the designer. A parameter is represented graphically as an input, except that the arrow pointing towards the node is replaced by a little circle. Values of parameters can be used by synthesis algorithms etc. The parameters of a node are usually assigned values when the node is used as a building block in the implementation of a more complex node. Since these nodes in turn may have parameters, such an assignment need not yet fix the value of the parameter. When a parameter gets a value, the buildin constraint propagation mechanism checks the correctness of all assertions afflicted by the parameter.

An example of an assertion is given below:

```
(== (rows (input A)) (length (input b)))
```

This assertion is used to specify that the number of elements of b should be equal to the number of rows of A. A specialized eval procedure has been defined which evaluates the assertions and builds a constraint network from them. This

procedure recognizes the selectors *input*, *output* and *parameter*, and adds the assertion to the property list of the corresponding input, output or parameter as a constraint on the value of the property.

The designer can also associate a behavior with a HIFI node, as discussed in the previous section.

Including the above in the definition of the Faddeeva node we would get the following expression:

```
(node-class (Faddeeva (:integer dimension))
  (inputs (:matrix A) (:vector b))
  (outputs (:vector x))
  (assertions (== (rows (input A)) (length (input b)))
              (== (parameter dimension) (rows (input A))))
  (behavior (make :vector
                  (qr-iter
                   (append (mapcar append
                                   (columns-of A)
                                   (columns-of (transpose b)))
                           (unity-mat (1+ (dimension A))))))))
```

Here we have added a selector *columns-of* to transform A from a matrix to a list of (column) vectors, so that we can use the procedure mapcar. The result of *qr-iter*, a list of numbers, is transformed into a vector, by the generic procedure *make*. All datatypes have a specific make procedure, their *maker*, that creates an object of its type using the auxiliary arguments of *make*. According to its first argument, *make* will dispatches to the appropriate *maker* procedure. We will assume that all objects are created by this generic *make*, thus making it possible to use generic procedures throughout.

3.2 Refinement

The next step is to define a dependence graph that gives an alternative specification of the behavior of the Faddeeva node via a refinement. We distinguish two forms of refinement:

1. explicit refinement
2. regular refinement

An explicit refinement specifies the behavior of a node in the form of a dependence graph. The graph consists of inputs, outputs, nodes and edges. The edges connect inputs and outputs of nodes. The inputs and outputs of the graph are the inputs and outputs of the node being refined.

A regular refinement will also result in a dependence graph. This time however the number of nodes is not known a priori, but instead given by a number, the iteration count. Also all nodes are of the same *node-class*, the so-called iteration node. The definition of a regular refinement amounts to the specification of an iteration node, an iteration count and a specification of the interconnections between the iteration nodes. Regular refinement is the HIFI analogon for tail-recursion.

For the Faddeeva algorithm the first step is an explicit refinement. This refinement separates the mapping at the input, from the actual qr decomposition. An explicit refinement is specified in the form of a (dependence) graph. The graph is entered into the HIFI system using the HIFI refinement-editor. This editor allows the

designer to graphically enter the definitions. A representation of the refinement is shown in Figure 9, in the form of the display shown to the designer when editing the refinement.

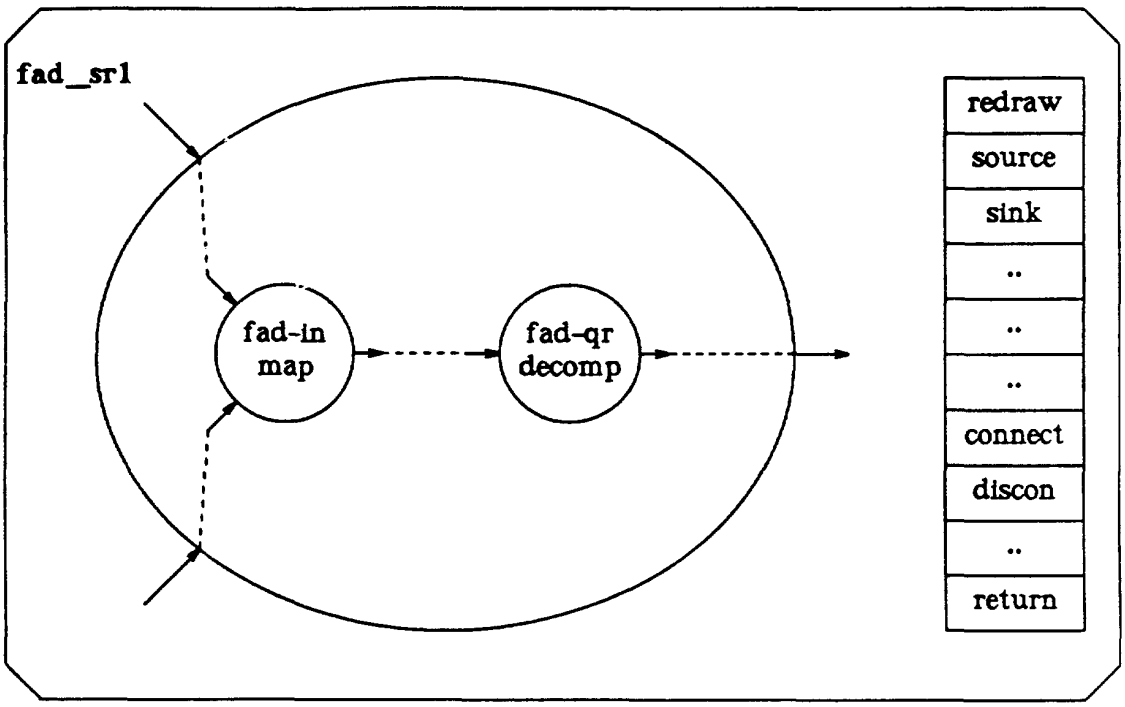


Figure 9. Faddeeva: first explicit refinement

In this case the graph consists of two nodes, that are instances of the node-classes fad-in-map and fad-qr-decomp. The definitions of these nodes are as follows:

```
(node-class (fad-in-map)
  (inputs (:matrix A) (:vector b))
  (outputs (:matrix A))
  (behavior (append (mapcar append
                            (columns-of A)
                            (columns-of (transpose b)))
                    (unity-mat (1+ (dimension A))))))

(node-class (fad-qr-decomp)
  (inputs (:matrix A))
  (outputs (:vector x))
  (behavior (qr-iter A)))
```

To simplify the example somewhat, we wil concentrate now on the fad-qr-decomp node. The next step then is the specification of a regular refinement of this node. Figure 10 shows the refinement after editing by the designer.
The designer specifies the number of iterations as an expression that may contain both parameters and properties of inputs and/or outputs. After editing the refinement it is interpreted by a procedure that converts the refinement to an internal representation. This procedure also checks the consistency of the refinement insofar this is not already done by the editor. If inconsistencies, e.g. due to a constraint mismatch are found, the refinement will not be put back.

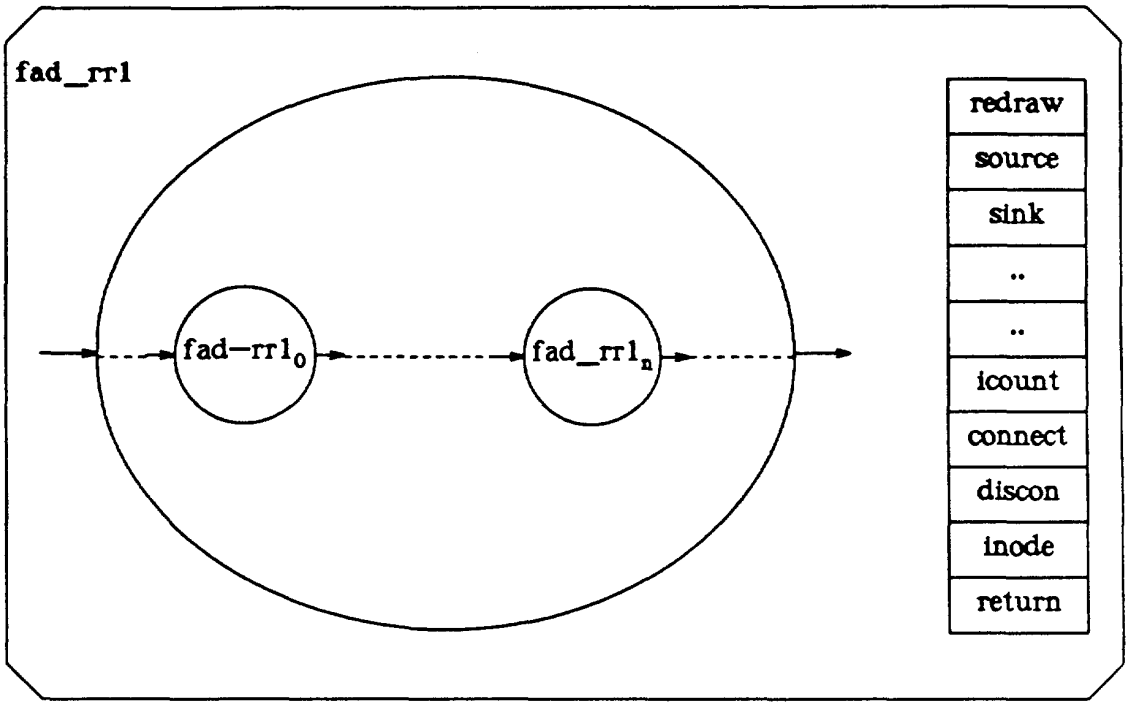


Figure 10. Faddeeva: first regular refinement

For now we wil assume that the node selected as the iteration unit for the regular refinement of the Faddeeva algorithm is called *fad-rr1*. Its initial definition could be something like:

```
(node-class (fad-rr1)
  (inputs (:matrix a))
  (outputs (:matrix a)))
```

The number of iterations of this node is given by the number of rows in the A matrix that serves as input to the node fad-qr-decomp, the node that was implemented through the regular refinement. The number of iterations can be specified as being equal to the number of rows in this matrix.

The third step is again an explicit refinement. This time the node *fad-rr1* is split in two nodes, corresponding to the vector and matrix reflect procedures in the original LISP procedure. The refinement is shown in Figure 11. With reference to Figure 11, we remark that the circle labeled *S* is a standard node, like sources and sinks. The S node splits a composite input value, e.g. a vector or a matrix, into its first component and the remaining components, much like the LISP procedures *car* and *cdr* operate on lists. In the HIFI system we also have *C* nodes. A C node takes an element and a composite value, and outputs a combination of the two. The behavior of these nodes is easily specified using generic selector and constructor procedures.

The fourth step is then the regular refinement of the matrix reflect node as an iteration of reflect operations on individual columns of a matrix. This refinement is shown in Figure 12. Here we have again S and C nodes. In the case of regular refinement, these nodes are implicitly repeated, as many times as necessary.

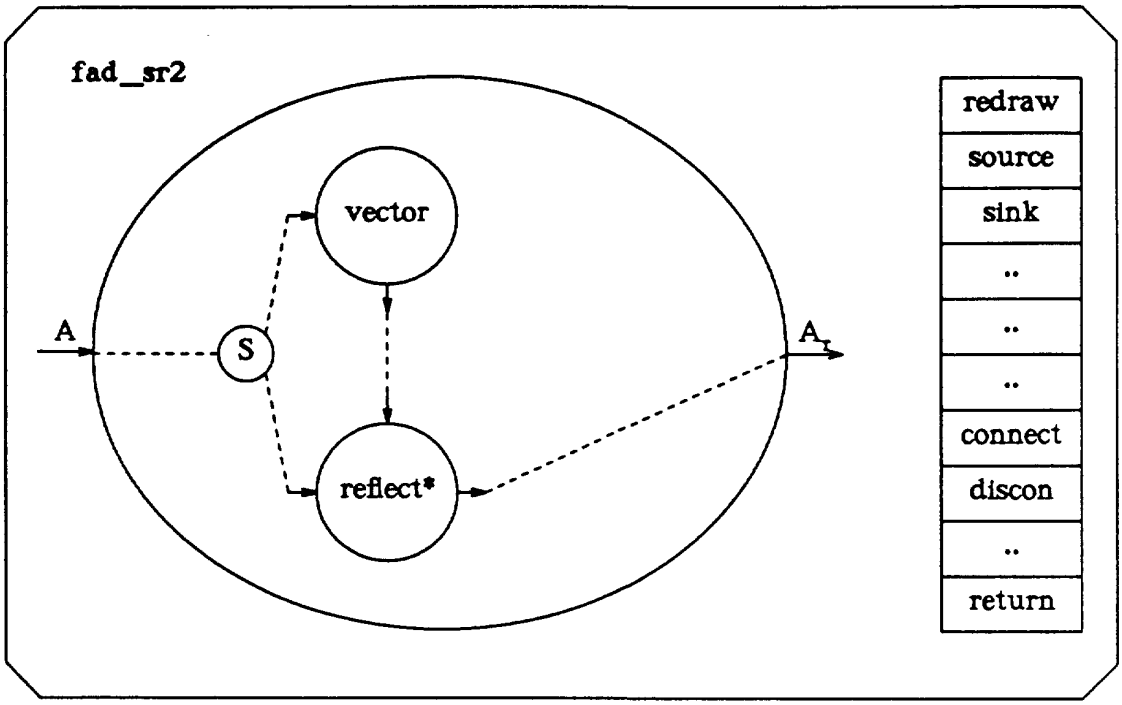


Figure 11. Faddeeva: Second explicit refinement

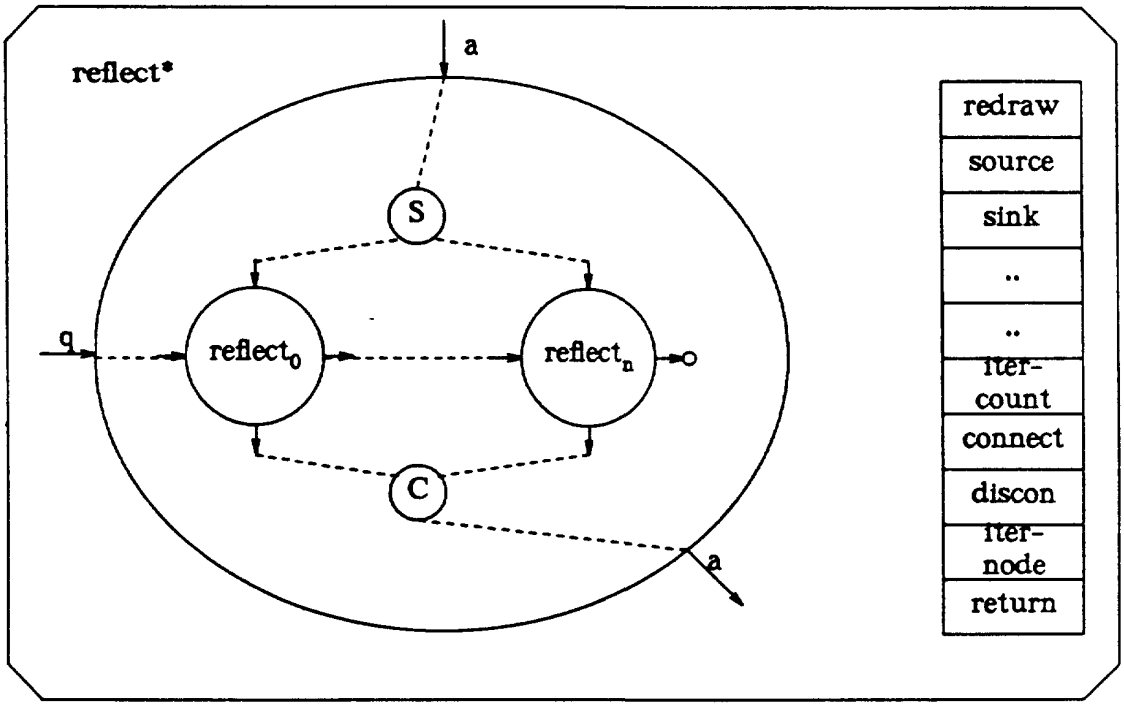


Figure 12. Faddeeva: Second regular refinement

It should be clear that by this time we have reached a DG representation of our original algorithm as shown in Figure 13. Although we have proceeded "by hand",

it should also be clear that this result could have been obtained automatically - see the discussion.

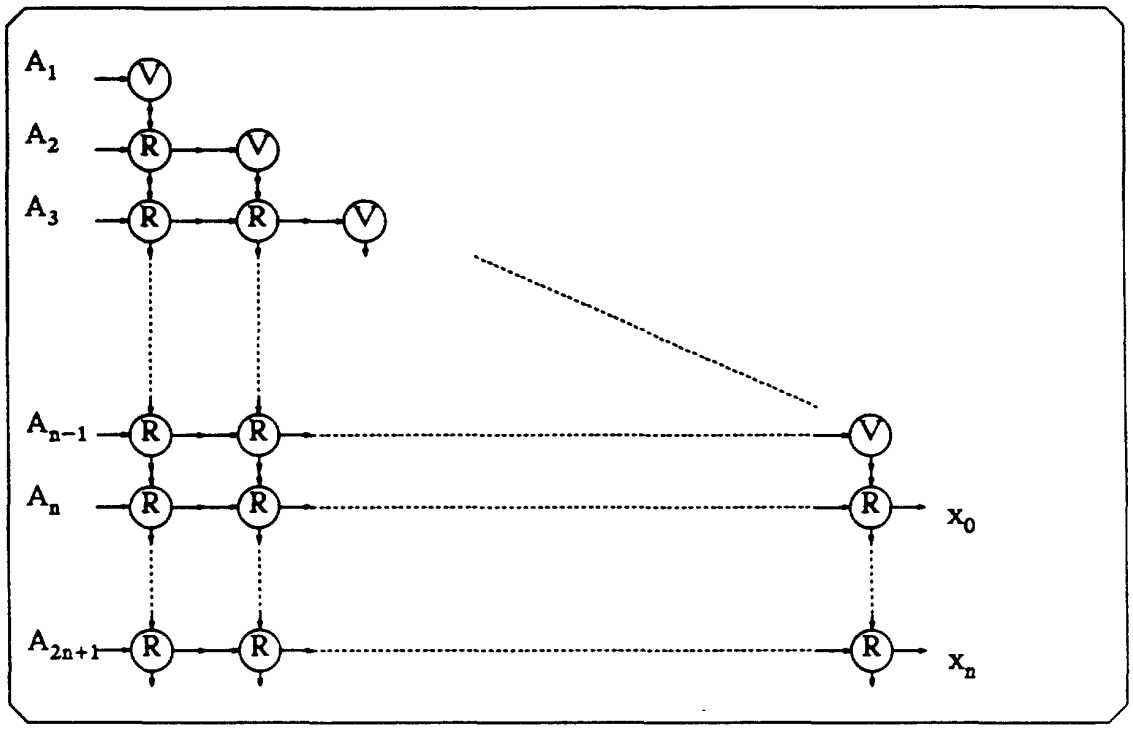


Figure 13. Faddeeva: Expanded Dependency Graphs

3.3 Mapping onto a processor array

The last part of the example discussed here is the mapping of one or more levels of refinement onto a processor array. At this point a partitioning technique must be chosen. For instance, the designer can start with "linear partitioning", in which he proceeds by choosing a processor space and a scheduling space. Subsequently, the nodes in the DG are mapped on the nodes in the processor space, thereby deriving the behavior of the processor nodes in the form of an AST definition. If the processor space is chosen horizontally, and the scheduling space vertically the partitioning results in Figure 14.

The definition of the VR nodes in Figure 14 is as follows:

```
(node-class (VR)
  (inputs (:vector A q) (:bit control))
  (outputs (:scalar r) (:vector A q) (:bit control))
  (initial-function control_0)
  (ast
   (node-function (control_0)
                  (inputs control)
                  (outputs control)
                  (next-function (if (= control 1)
                                     vector
                                     reflect))
                  (behavior (list control)))
   (node-function (control_1)
```

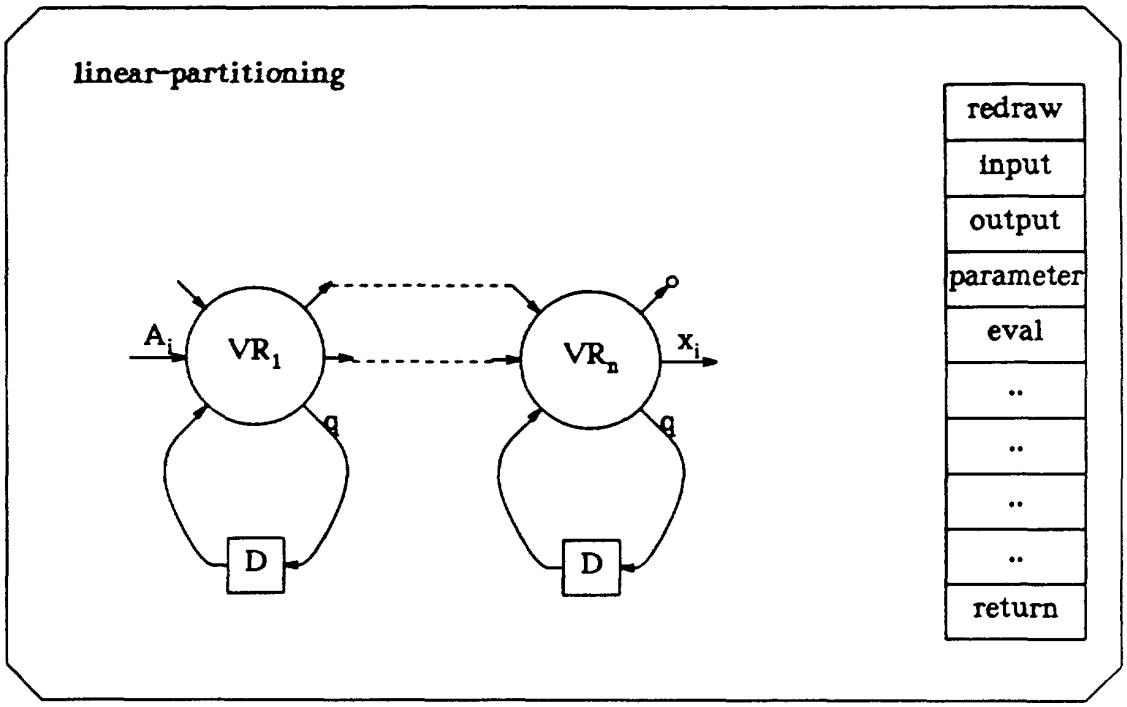


Figure 14. Faddeeva: Partitioned SFG

```
                (inputs control)
                (next-function (if (= control 1)
                                   vector
                                   reflect))
                (behavior (list control)))
  (node-function (vector)
                (inputs A)
                (outputs q r)
                (next-function control_1)
                (node-class vector))
  (node-function (reflect)
                (inputs A q)
                (outputs A q r)
                (next-function control_0)
                (node-class reflect))))
```

The data state appearing as a delay edge in Figure 14 is defined as follows:

```
(connect (formal-arg 'q (outputs-of VR))
         (formal-arg 'q (inputs-of VR))
         (:vector)
         (delay 1))
```

The VR nodes are controlled by a control bit that is propagated along the array. By doing so, all VR nodes can be exactly equal, which will simplify further partitioning. If we for example had defined the VR nodes using a clock as in the square-root example, the position of the nodes would have to be made explicit. This would disturb the regularity, and complicate the partitioning. Note that the reduction in the number of columns along the array, is now quite naturally encoded, by not propagating the control bit in state *control_1*.

Further partitioning is necessary if the available resources do not match (are smaller than) the number of components in the SFG.

Given a 'regular', 'unidirectional' SFG two partitioning strategies are possible [13, 14].

1. LSGP, Local Sequential Global Parallel
2. LPGS, Local Parallel Global Sequential

A combination of the two strategies allows one to control the ratio of local and global memory.

An example of the LPGS partitioning strategy is shown in Figure 15, for the case in which pN = n, where N is the number of processors in the (partitioned) array, n is the row dimension of the A matrix, and p is an integer that denotes the number of global iterations, the so-called *global iteration index*. In case n is not evenly divisible by N, the A matrix should be properly extended.

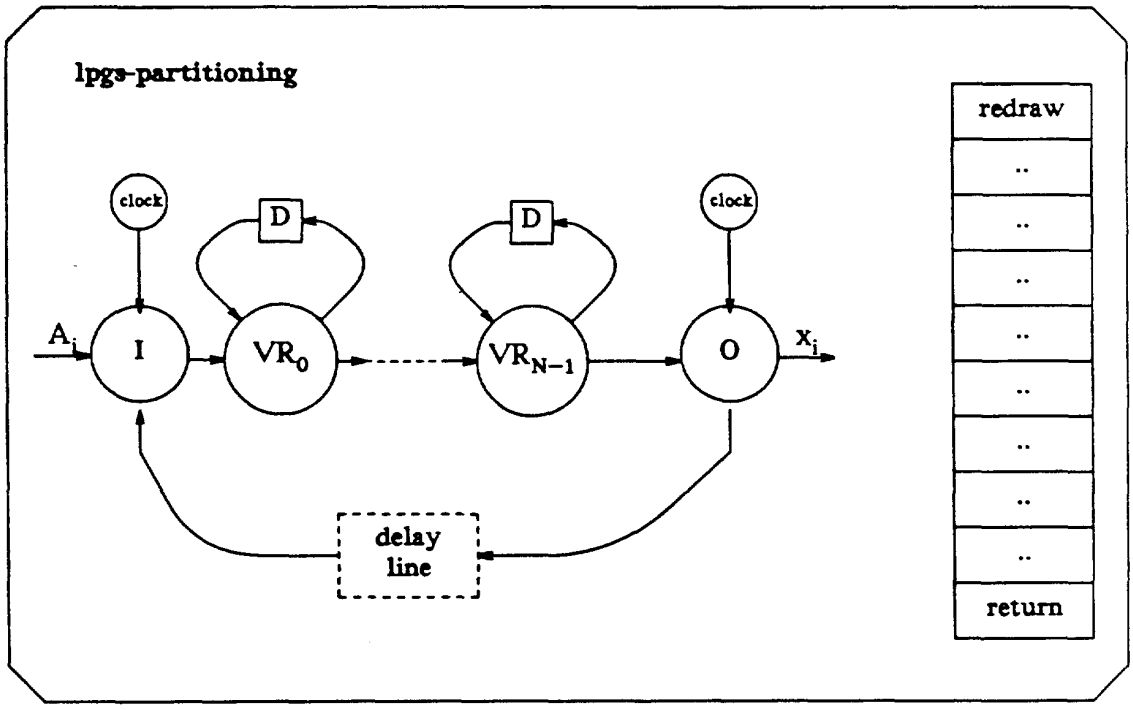


Figure 15. Faddeeva: LPGS partitioning

The switching network associated with this partitioning, denoted by the dashed box labeled *delay line* in Figure 15, is shown in more detail in Figure 16. In fact it implements a type of FIFO buffer. The input and output switches (*IS*, *OS*), except the rightmost ones, are programmed so that they switch after the first N tokens, where N is the size of the processor array. The two rightmost ones, resp. IS_p and OS_p switch after n + 1 tokens. The output switches first pass their data down, and then pass it to the left. The input switches first pass data from up to right, then continue passing data from left to right. The final token resets all switches to their initial state. This can be accomplished by means of an additional control bit.

The *IS* and *OS* nodes in Figure 15 are controlled by the global iteration index, which is represented by a clock. The *IS* nodes pass tokens from left to right during the first iteration. Subsequently it passes data from down to right. The *OS* nodes pass tokens from the left down, except for the final iteration, when data is outputted to the right. Both, the *IS* and *OS* nodes are reset to their initial state after the final iteration.

For clarity the bits controlling the operation of the VR nodes and the switches are not shown in Figure 15. It is interesting to note that the partitioning also applies to the propagation of the control bits.

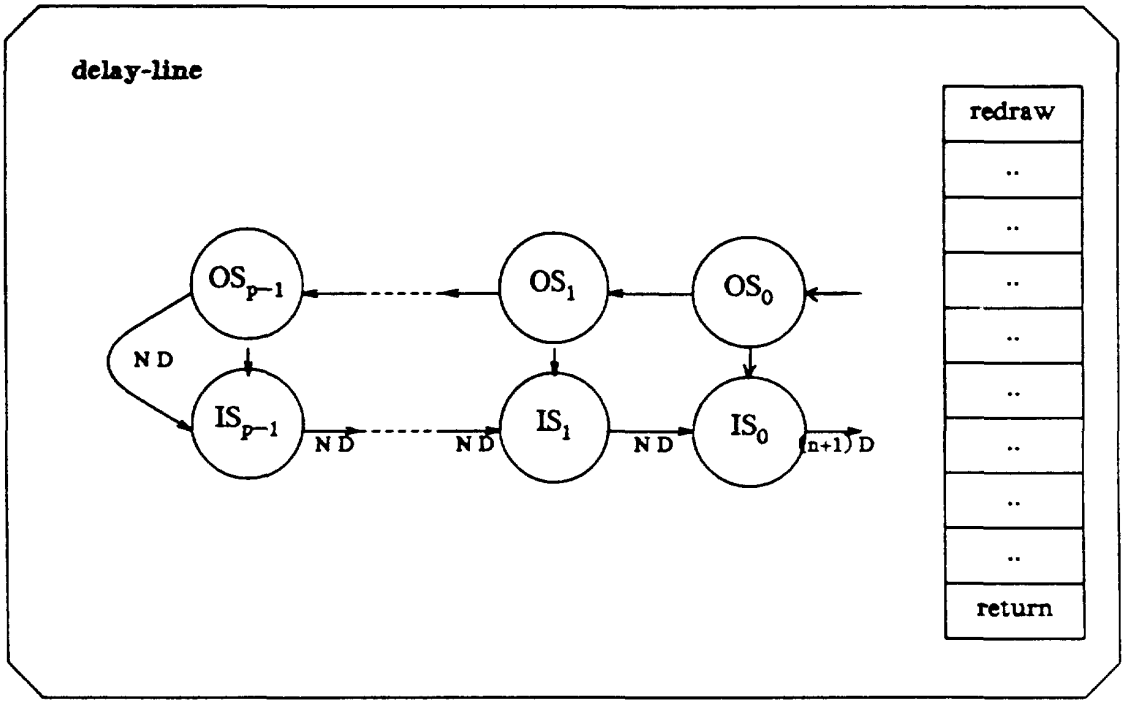


Figure 16. Faddeeva: LPGS partitioning - delay line

Having reached a satisfactory partitioning of the Householder algorithm, it is possible to proceed further and to decompose the Householder processor into a sequence of Givens rotations. The resulting architecture now becomes two dimensional, and it will be necessary to move back to a (now three-dimensional) DG, which will provide the basis for further partitioning. Proceeding further, more or less like in the Householder case, we can perform partitioning of the LSGP/LPGS type in the horizontal and vertical directions. This leads to very attractive systems in which the algorithm is completely implemented with dedicated hardware, and the switching network around it contains a minimal number of control functions. Presenting these results here would exceed the scope of the paper. We refer to recent publications of some of the authors: [29,26]

4. DISCUSSION

In the preceeding pages a number of issues have been raised to which we devote some more attention in this section.

4.1 Single assignment and automatic design

First of all, there is the question of compilation of the original code into single assignment form. See e.g. the code of the Faddeeva algorithm early in section three. Because of the repeated calls to the procedure *qr-iter*, a number of variables with the same name - e.g. "a" and "q" - will be duplicated and represent different instances. Hence the path from the original algorithm to the dependence graph will have to make these variables explicit. Doing this is in fact equivalent to producing single assignment statements as one would do when faced with a Fortran description of the same algorithm. There are however some major differences. In contrast to the Fortran code, the LISP code gives detailed information on the transformations (functions) that are forced on the data during operations. The hiding of actual values is done not because they are (conceptually) overwritten, but because a number of operations (grouped together) are formally equivalent. When parsing the code, the LISP interpreter will force the creation of a sequence of new environments to which instances of variables and of operations will be attached [15]. The LISP interpreter generates, in effect, the dependence graph. As a consequence the refinement phase of the design procedure can be made automatic: what was done by hand in section three can actually be done by interpreting the LISP code. The only decisions left to the designer are choosing the characteristics of a partitioning step. All the rest: structure as described in the SFG, scheduling etc. can be generated automatically. Since the partitioning parameters do not enter the design explicitly (although they greatly influence the performance !) we obtain a design that is "Correct by construction".

4.2 Timing Verification

In the present paper, the question of timing verification has not been considered. However, it should be clear that the formalism developed so far will allow for it. Once a SFG is correct (in the sense of section two), timing can be assigned to the propagation on edges and nodes. Timing constraints can be added and verified (e.g. all tokens must be present at relevant inputs of a node, when a clock-token arrives). In this way, the asynchronous single-token pass machine can be converted into synchronous machines. The timing constraints can then be further propagated down the design tree and verified at lower levels. This topic will be the subject of another paper.

4.3 On the model used

The HIFI computational model has been tested on a variety of examples. This has brought some possible problems to light. The main objection seems to be that the model does not allow routing decisions on the basis of inputs. An example may clarify the issue. With reference to Figure 17, suppose that d is an (n + 1) bit word, with d_0 a control bit, with the following meaning: if d arrives, and d_0 is "0", then the node has to execute f_1, which takes bits 1 .. n of d (d_r) as input, discards state, and outputs state+.
On the other hand, if $d_0 = 1$, then f_2 is executed, which takes d_r and state as input, and outputs out and state+. Switching between f_1 and f_2 is not outside our model,

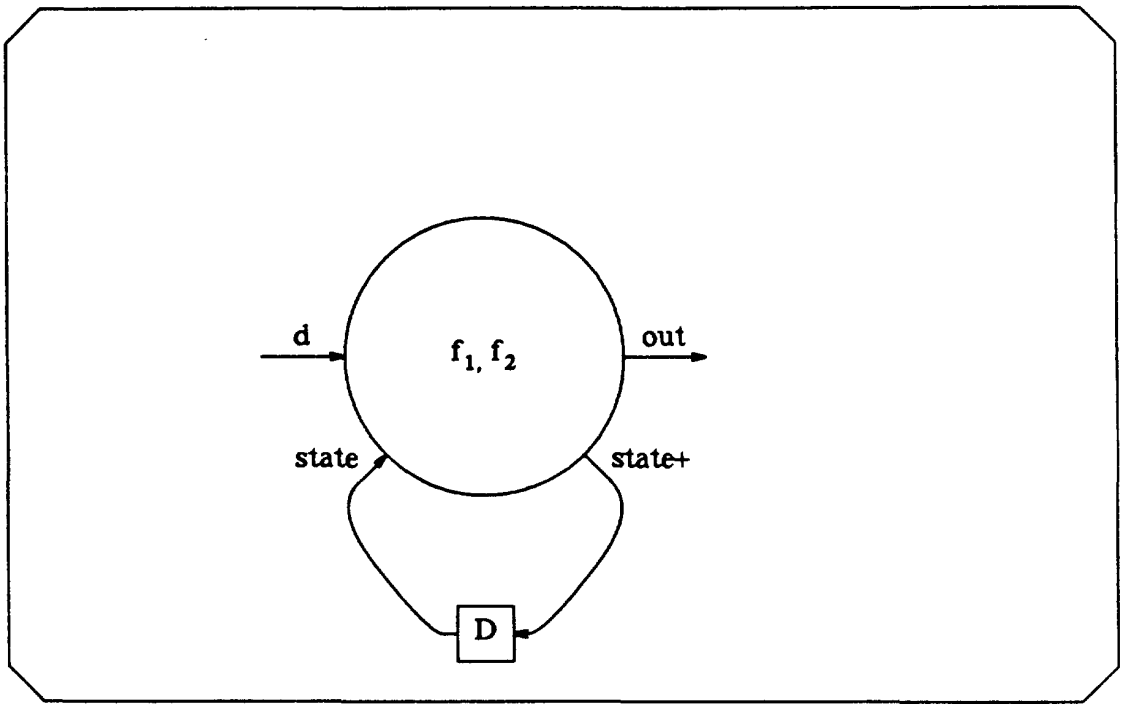


Figure 17. Example of a system that falls outside HIFI

but having f_1 and f_2 handling different inputs and outputs is. A correct HIFI representation for this case is shown in Figure 18 and illustrates the properties of the model.

The initial function of the node now is f_i, which takes a token from d_0 and then decides between f_1 and f_2. F_1 will consume a token from d_r and generate one on state+, after which it will revert to f_i. F_2 will consume a token from state and d_r and generate tokens on out and state+. The switching action of d_0 becomes explicit in the model. In an actual electronic circuit it will take some time to set up f_1 (or f_2). With a good estimate on this time, attached to f_i, we can actually verufy the timing already at this level of the design. Incidentally, we see that the store representing D has to be operated dynamically (which will almost certainly be the case in the actual circuitry). The question could next be raised whether we can represent as complex a circuit as a RAM. In the present model this can be done as follows:

The F(RAM) function is dependent on the content of the RAM (address → data) It takes as inputs {C, address, IN} and generates OUT. Depending on the mode coined by C (read or write), F(RAM) will either modify itself to F(new-RAM) or stay stable as F(RAM). In all cases it will consume input tokens and produce an output token. The controller node starts from f_{in} which interprets {C, address} solely, then switches to f_{read} or f_{write} which will take care of the token dispatch, just as in the previous example (f_{read} has no inputs, and outputs C, address and (dummy) IN after which it branches to f_{out} which takes input OUT and outputs OUT and branches to f_{in}; f_{write} takes input IN, produces outputs C, address and IN, and branches to f_{dis} which in turn simply discards OUT and branches back to f_{in}).

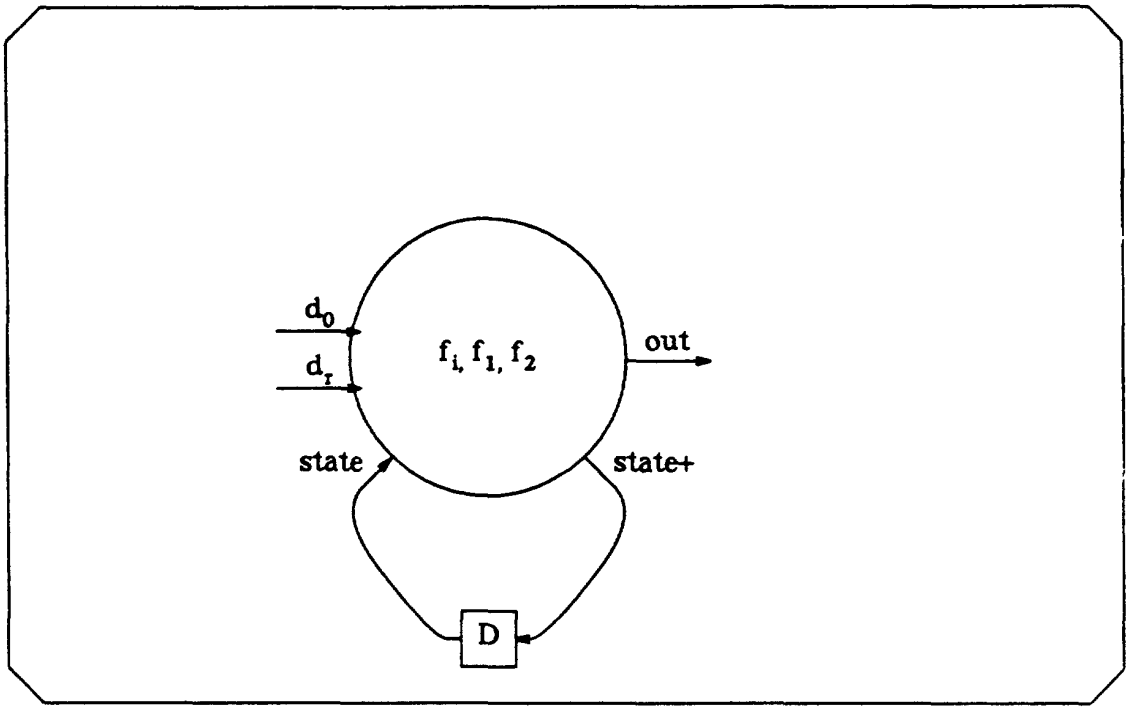


Figure 18. HIFI solution for the problem of Figure 17

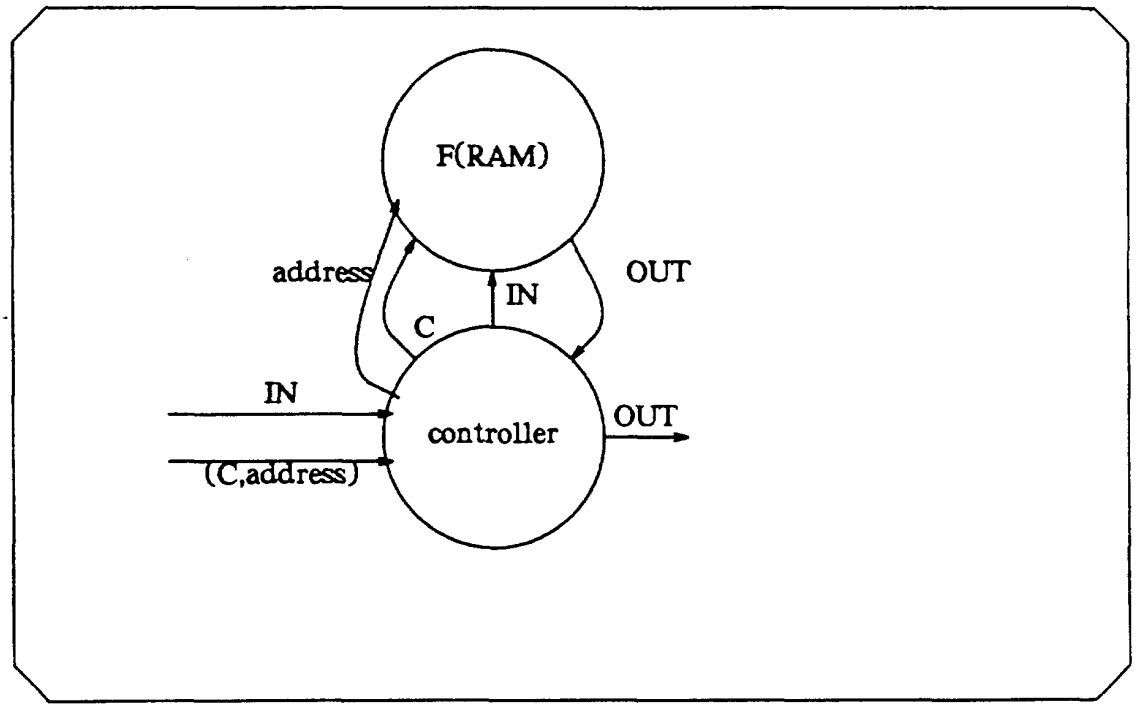


Figure 19. Representation of a RAM

The example shows that the present model is sufficiently powerful to represent major standard functions. It shows also that the designer is forced by the model to

make relevant functions (state transitions) explicit. The single token-pass discipline does not allow for hidden effects, especially concerning the routing of data. In the RAM example, the controller is forced to dispatch the data explicitly, after having interpreted the (dataflow) control bit C. It could also have been handled more implicitly whereby the controller would interpret the control bit and decide further routing on the basis of it. We have decided not to allow such implicit routing: it would make the state of the SFG at a given event dependent on the data at that event, and hence would make it dependent on the functions actually being executed by the nodes. Timing verification would become very difficult. Although the model is powerful enough to handle all deterministic situations, it does put restrictions on the designer, who is forced by the model to make design decisions explicit at every level of the design.

As a conclusion, we would like to emphasize that the Signal Flow Graph description that results from a design procedure is intended to represent a concrete system. It is not purposed to be yet another hardware description language, but a concrete representation of the system from which a silicon compiler can take off. The elegant description of the partitionings obtained in section three should be sufficient proof of the power of the method.

References

1. G. Goosens, "An Optimal and Flexible Delay Management Technique for VLSI," pp. 409-418 in *Computational and Combinatorial Methods in Systems Theory*, ed. C.I. Byrnes, A. Lindquist, North-Holland ().

2. J.Backus, "Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs," *Comm. ACM* 21 pp. 613-641 (August 1978).

3. A. Cremers and T. Hibbard, "Executable Specification of Concurrent Algorithms in terms of Applicative Dataspace Notation," in *VLSI and Modern Signal Processing*, ed. S.Y. Kung, H.J. Whitehouse, T. Kailath, Prentice Hall (1985).

4. R. Milner, "Flowgraphs and Flow Algebras," *Journal of the ACM* 26(4) pp. 794-818 (Oct. 1979).

5. A. Izawa and T.L. Kunii, "Graph Based Design Specification of Parallel Computation," *Techn. Rept. Dept. of Information Science, Univ. of Tokyo*, (Dec. 1983).

6. E.A. Ashcroft and W.W. Wadge, "Lucid: A Non-Procedural Language with Iteration," *Comm. of the ACM*, pp. 519-526 (July 1977).

7. P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier, "SIGNAL: A Data-Flow Oriented Language for Signal Processing," Publication # 246, IRISA , Rennes, France (January 1985).

8. J. Annevelink, "A Hierarchical Design System for VLSI Implementation of Signal Processing Algorithms," *Proc. Intl. Conf. on Mathematical Theory of Networks and Systems MTNS-85*, (June 1985).

9. P. Dewilde, E.F. Deprettere, and R. Nouta, "Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms," in *VLSI and Modern Signal Processing*, ed. S.Y. Kung, H.J. Whitehouse, T. Kailath, Prentice Hall (1985).

10. C.V. Ramamoorthy, K.M. Chandy, and M.J. Gonzalez, "Optimal scheduling strategies in a multiprocessor system," *IEEE Trans. on Computers* 21(2)(Feb. 1972).

11. Sailesh K. Rao, "Regular Iterative Algorithms and their Implementations on Processor Arrays," Ph.D. Thesis, Information Systems Lab, Stanford, California (Oct. 1985).

12. D. Moldovan and J.A.B. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," *IEEE Trans. on Computers* C-35(1) pp. 1-12 (Jan. 1986).

13. K. Jainandunsing, "Optimal Partitioning Schemes for Wavefront/Systolic Array Processors," *Proc. IEEE Intl. Conf. on Circuits and Systems*, pp. 940-943 (May 1986).

14. H. Nelis, K. Jainandunsing, and Ed. F. Deprettere, "A Systematic Method for Mapping Algorithms of Arbitrarily Large Dimensions on to Fixed Size Systolic Arrays," *Proceedings ISCAS-87 (to appear)*, ().

15. H. Abelson and G. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press (1985).

16. T. Kailath, *Modern Signal Processing*, Springer Verlag (1985).

17. S.Y. Kung, H.J. Whitehouse, and T. Kailath, *VLSI and Modern Signal Processing*, Prentice Hall, New Jersey (1985).

18. H.T. Kung, B. Sproull, and G. Steele, *VLSI Systems and Computations*, Computer Science Press (1981).

19. K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, Mc. GRaw Hill (1984).

20. K. Jainandunsing and Ed. F. Deprettere, "Design and VLSI Implementation of a Concurrent Solver for N Coupled Least Squares Fitting Problems," *IEEE Journal on Selected Areas In Communication* SAC-4(1) pp. 39-48 (Jan. 1986).

21. R.W. Floyd, "Assigning Meanings to Programs," *Proceedings of Symposia in Applied Mathematics 19*, pp. 19-32 (1967).

22. S.Y. Kung, "VLSI Array Processors," *Intl. Workshop on Systolic Arrays*, (July 1986).

23. S.Y. Kung, *VLSI Array Processors*, (to appear) Prentice Hall (1986).

24. H. Nelis, K. Jainandunsing, and Ed. F. Deprettere, "Automatic Design and Partitioning of Systolic Arrays," *Tech. Report, Dept. of EE, Delft Univ. of Technology*, (August 1986).

25. E.W. Dijkstra, "Guarded Commands, nondeterminacy, and formal derivations of programs," *Comm ACM* 18(8) pp. 453-457 (August 1975).

26. K. Jainandunsing and Ed. F. Deprettere, "A novel VLSI System of Linear Equations Solver for real-time Signal Processing," *SPIE Symp. on Optical and Optoelectronic Applied Science and Engineering*, (Aug. 1986).

27. V.N. Faddeeva, *Computational Methods in Linear Algebra*, Dover publ. , New York (1959).

28. J.H. Wilkinson and C. Reinsch, *Linear Algebra*, Springer Verlag, New York (1971).

29. E.F. Deprettere, P. Dewilde, and R. Udo, "Pipelined cordic architectures for fast VLSI filtering and array processing," *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pp. 41A6.1-41A6.4 (March 1984).

INTELLIGENT VLSI DATAMANAGEMENT

P. Dewilde, J. Annevelink, R. van Leuken and P. van der Wolf
Delft University of Technology
Delft, the Netherlands

Abstract

With the field of Computer Aided Circuit Design moving into the realm of "intelligent systems", the logical representation of design data becomes of the utmost importance. The present paper starts out with showing how the multiview and hierarchical structure of the VLSI design data can explicitely be exploited to increase design efficiency. It soon becomes obvious that a versatile datamodel is needed to represent the structure of the data. We define a semantic datamodel and show how it is applied to produce a useful schema. Finally, we show how an object oriented design data manager is constructed and how it is capable of satisfying the requirements of (interactive) efficiency and of transparent handling of the data at a high level of abstraction.

1. Introduction

We consider the problem of managing the VLSI design data so that the following requirements are satisfied:

1. Explicit use is made of the structure of the data, i.e. views and hierarchy are exploited to reduce the complexity.
2. The system is build in a modular and additive way, in particular data management and design functions are independent and non-redundant. A major problem with many present day systems is the necessicity of frequent reconversion of data. The programmers should be given a standard view on the data independently of the application at hand.
3. The system must support fast interactive handling of the data, even at higher levels of abstraction. E.g. merely defining a higher level description language will not do.
4. Support has to be given to design evolution, top-down as well as bottom-up design, and (controlled) multiuser access. In particular, version control must be supported (in a way that is compatible with the hierarchy).
5. The system must be designed in such a way that expert tools can be added. An "object oriented" style will do in particular.

The requirements detailed above are surely not easy to satisfy all together. Generally applicable databases have unsuitable datatypes, are rigid and slow. On the other side, functional or logic languages like LISP [1] or PROLOG [2] are too unspecific and tend also to be extremely slow. A major problem with them is the poor control the application programmer has on the localization of the design data.

Many classical design tools like Design Rule Checkers (DRC's) or Extractors while efficient and not uncompatible with the design hierarchy, do not exploit that hierarchical structure in order to achieve theoretical efficiency. Without due care to the latter aspect, it may not even be possible to check a large repetitive structure like a large memory array.

In the following sections we shall describe solutions to the following basic three questions:

1. How to exploit the multiview/hierarchical structure of the data to achieve efficient verification.

2. How to model the data so that it includes in a logical way the multiview and hierarchical structure, independence of data and functions, version control and multiuser access.

3. How to implement the datamanager so that at the same time fast (interactive) design data manipulations and intelligent (expert) representations are supported, and on the other hand design tools can retrieve large amounts of data.

The results presented in this paper have been arrived at after extensive study of the litterature. The special datamanagement properties needed for VLSI design and CAD in general were well recognized by Katz [3] . The relational datamodel presented earlier in the literature, can easily be shown to be inadequate. Especially the problem of version control in a hierarchical environment is not easily solved. A datamodel that is much better suited for design work is the functional datamodel first proposed in Daplex [4]. A recent contribution in the field is e.g. DODM [5] . A description of different approaches to functional and semantic datamodels can also be found in that reference. From ter Bekke [6] we borrowed the principle of equivalence between an object and its attributes. Many present proposals for semantic datamanagement are based solely on fixed types and are too restrictive and also unnatural because the more flexible notion of "set" is fundamental in Mathematics as well as in Database work. A major step forward in practical and intelligent representations of abstractions has been made by the creation of Smalltalk [7] . It takes the point of view of procedural semantics, and its concept of "objects" meshes nicely with the concept of "frames" dear to artificial intelligence [8, 9] . In recent times, object oriented extensions of classical languages like C or LISP have been proposed. Together with the use of "generalized datatypes" and the inheritance mechanisms connected with them, a very powerful environment can be created that allows for natural representation of the design data at a high level of abstraction, yet without preventing efficient datahandling. Good references for this circle of ideas are [10, 11] . Finally, a different line of approach, yet with equally attractive features, is "functional programming" which was proposed in [12] . We borrow from it the concept of "algebra of programs" which in CACD proves to be very useful, especially to describe designs and design refinements [13] .

In the present paper we shall not attempt to make an exhaustive presentation of all the new concepts mentioned above. We shall take a rather inductive approach: from the multiview/hierarchical design problem onwards to the abstractions needed for intelligent representation of the data.

2. The Multiview/Hierarchical Structure of the Design Data

It is one thing to say that a design system stores data in a hierarchical way and quite another to actually have design tools exploit that structure to achieve maximal efficiency. In this section we shall show how the latter can be done. To fix ideas we shall use the problem of low level design verification (Design Rule Checking and Extraction) to illustrate the concepts. Other areas where hierarchy plays an important role are: algorithmic and system design, placement and routing, silicon compilation. In all the areas cited there exist now "classical" tools. Yet with the advent of smaller dimensions (micron or even submicron), the electrical verification of still more complex circuits becomes even more critical, and there is a need for tools that achieve indeed minimal complexity.

Suppose that your hierarchical definition at a certain level - called cell -, contains a subcell that is repeated a large number of times in a square array (fig. 1 - this situation happens many times in any single VLSI design).

The interaction between the subcells belonging to the array should be checked only once as indicated in fig 1. The checking program should recognize such a situation. How this can be done precisely is to be found in [14, 15] , and will briefly be described here.

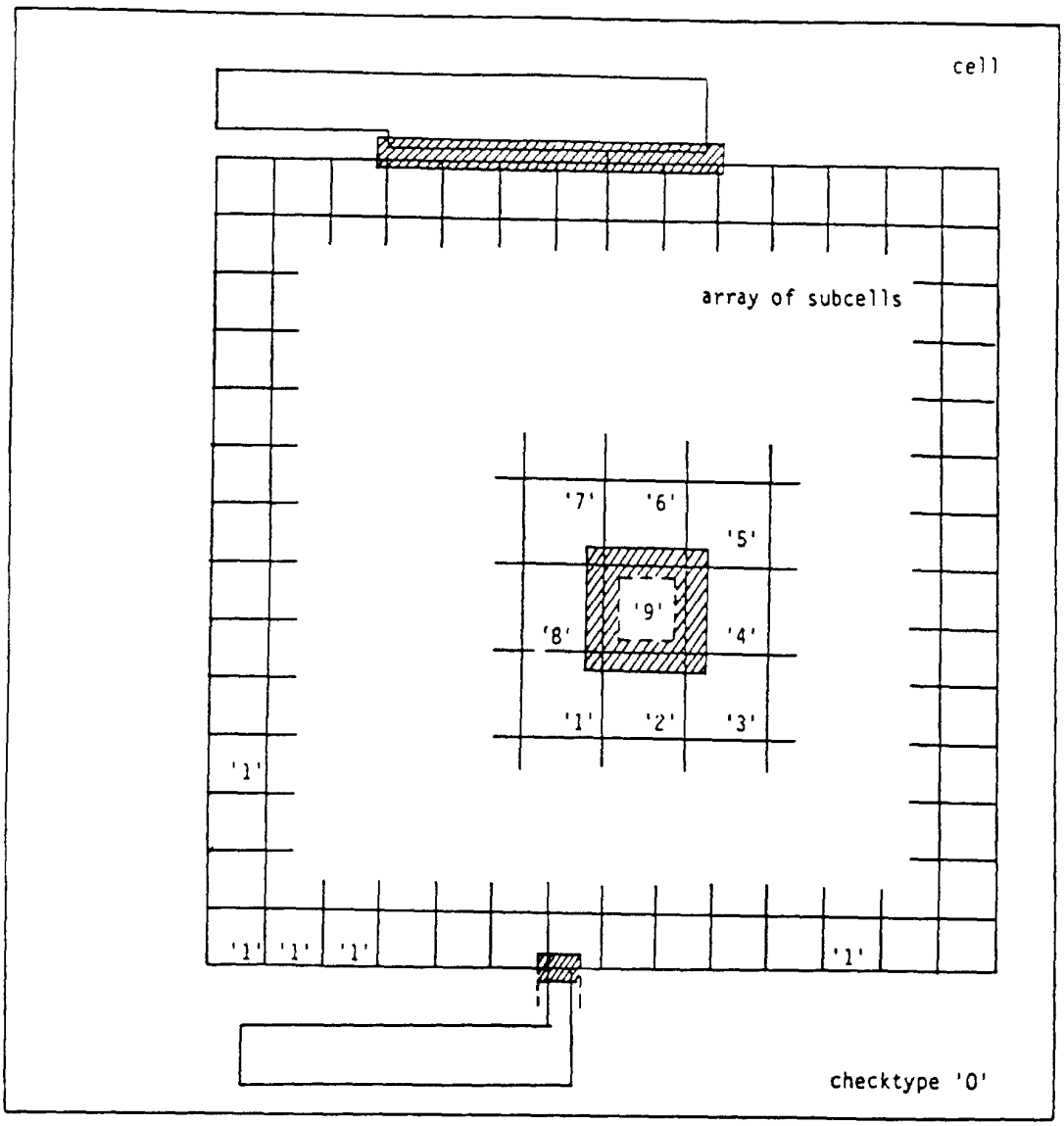


Figure 1. The subcell being repeated in an array in cell should be checked only once against its neighbours

Let us consider the problem of checking "cell" at its own hierarchical level, i.e. presuming that "subcell" (and all its subhierarchy) are already checked and are proven correct. We have of course to check all the original primitives of "cell" with respect to each other: that is classical DRC. The problem is the checking of the primitives of cell w.r. to the primitives belonging to its subcells in such a way that (1) all errors are detected, (2) no fake errors are reported (see further for an example) and (3) redundancies are avoided. A geometrical region where subcell primitives can influence the higher level is called an "active region" - it occurs at the boundaries of cell and its subcell as well as between subcells. At this point two strategies are possible:

1. The critical instances of the subcells are evaluated, and the primitives of "cell" which belong to the respective active regions are attached to the instances, which are then passed to the DRC.
2. For all the critical instances of the subcells, the primitives of the subcell that belong to the active region are promoted to "cell" with an indication of their status, in the form of "checktype" information.

While both approaches are technically correct, we prefer the second one because (1) it turns out to be more efficient: only one pass through a DRC is needed and (2) it has logic appeal: it is an instance of the following general principle of hierarchical verification:

> Any primitive in the subhierarchy that may have a non-local effect on the top level has to be instantiated at that level.

Fig. 2 illustrates the issue for the case of DRC. In the case of extract the situation is much the same, e.g. what must happen when a transistor originates from the intersection of a primitive at one level of the hierarchy with one at another?

The primitives of "cell" together with selected primitives of subcells which have a direct effect on "cell" is called the "augmented instance" of "cell". It is important to attach the checktype information to the promoted primitives so that no unnecessary checks are made e.g. of elements belonging to the same subcell. "Cell" will be correct if (1) the design rules for all the primitives of level '0' are satisfied and (2) the design rules are satisfied for primitives of unequal checktype. This seems good enough except for possible fake errors due to connectivity. In fig. 3 such a situation between cell and subcell is depicted.

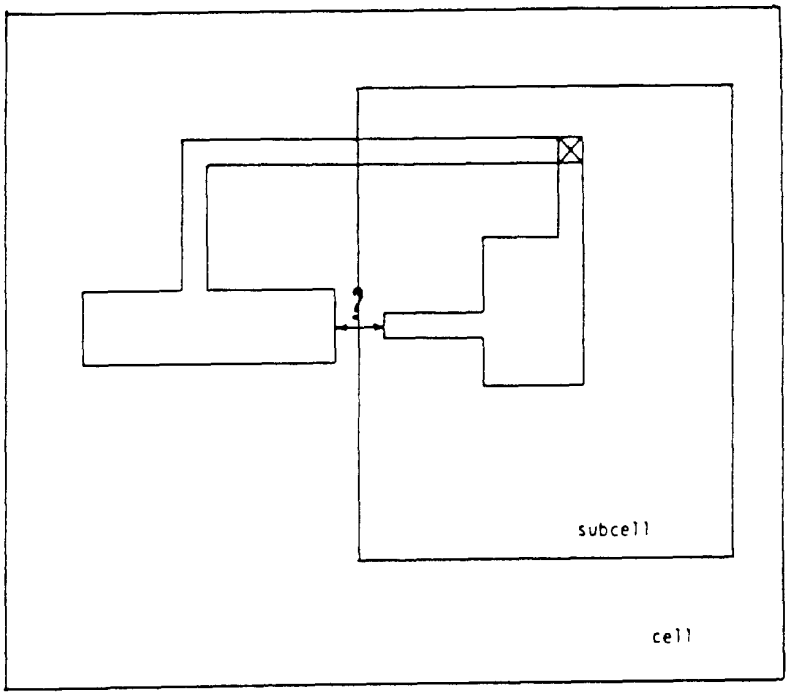


Figure 3. Fake error due to connectivity

Although such fake errors seem trivial, they tend to occur very often especially in repetitive designs, and the system should be able to remove them. Connectivity clearly is a global way of interaction between primitives which cuts through the hierarchy. A solution to the problem has been presented in [16] . The method uses what is called a "State-Ruler-Scan": it consists of a left-to-right scan of the layout primitives (edges) as was originally proposed in [17] with the necessary global information attached to the field as state information. In the case of design rule checking the only extra information needed beside the checktypes is that of connectivity which is attached to the scan as a "group number". Because this information is already propagated through the subcells (their terminals have also group information), fake errors due to connectivity can be avoided.

From this section it should be clear that it is indeed possible to exploit the hierarchy directly to reduce the complexity of the verification of the lay-out. For more information on this subject, e.g. the use of the method for lay-out-to circuit extraction or the accomodation of more complex design rules, see [18] .

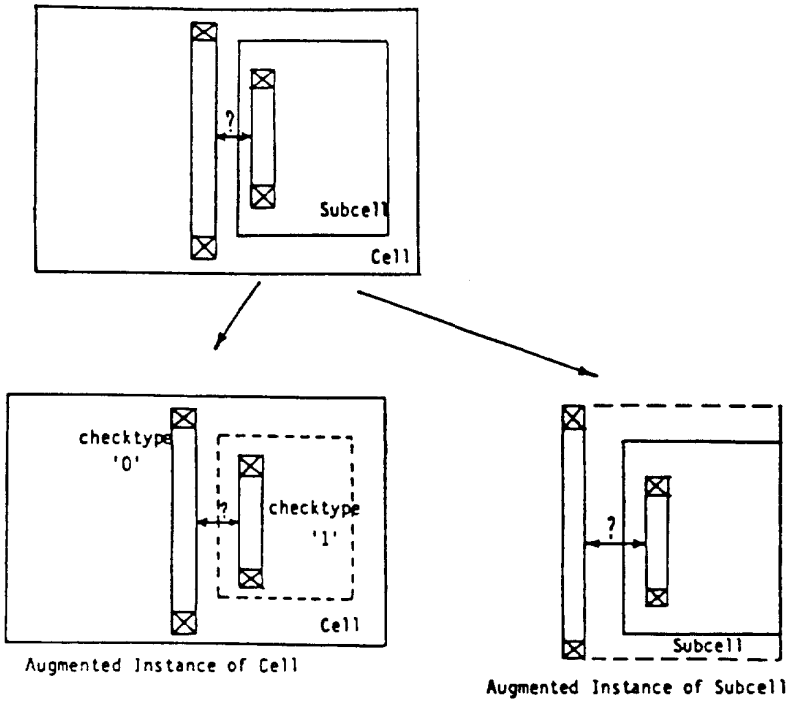


Figure 2. Deciding the question: for checking hierarchies does one promote active elements of the subtree, or does one demote active elements of the parent?

Other systems have been proposed in the litterature, see e.g. [19] . The system proposed here has the advantage that it achieves minimal complexity both in the exploitation of the hierarchy and in the scan procedure (proportional to the number of edges). Because of the need for systematic use of the hierarchical information, the next topic that we shall consider is that of efficient datamanagement.

3. Hierarchical Multiview Datamanagement

The tricky datamanagement issue! There is no topic in CACD (or in general CAD) where good systems and solutions are so scarce and problems loom so large. The original reaction of the (naive) designer is to plainly ignore the problem. However, anybody who has seriously tried to set up a system that has to manage complex design data has been forced to face it. Moving into the realm of "intelligent systems", data representation issues become crucial.

Database Principles

The accepted way of presenting the structure of a database is by showing a *dataschema* expressed in a *datamodel*. These basic notions we quickly review:

1. A *Datamodel* is the collection of mathematical concepts used to describe the structure of the datarepresentation.
A classical datamodel is the "relational" in which all data is represented as belonging to relations between sets of primitive data.

2. A *Dataschema* shows the representation of the data in a specific environment using the datamodel choosen.

3. A *Database* is the concrete implementation of a schema in a given case.

Modern datamanagement systems have been moving up in level of definition. This is highly desirable from an implementers point of view provided efficiency is maintained, because the more abstract levels will necessitate much less code, will be more understandable, less prone to mistakes and better maintainable.

In the course of our own work in the area [18] it has been become clear that the best datamodel for the design situation is the semantic. Instead of diving into database considerations, let us just work inductively and construct a semantic dataschema for our application.

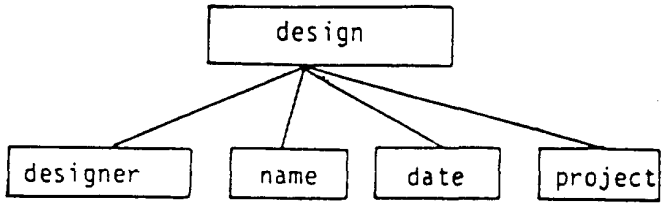
The Semantic Datamodel in the Design Situation

Litterature: see DAPLEX [4] SMALLTALK [7] and further [20, 5, 6] . Semantic datamodels use two types of abstractions: *aggregations* and *generalizations* (with inverses respect. *decomposition into attributes* and *specialisation*. They are shown schematically in fig.4.
Each block in fig.4 represents a class of objects which are characterized by their attributes. A semantic dataschema must satisfy the basic requirement of *invertibility* which states that a given object is equivalent to the collection of its attributes. In a similar vein specializations of an object inherit its attributes (with additional attributes added).

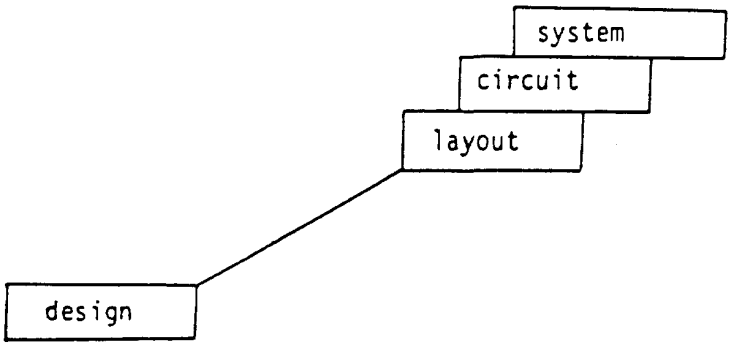
Objects as defined in Smalltalk [7] additionally possess a collection of methods which also help to characterize them and which define what kind of operations can be executed on them, and how. Relations between such objects can again be arranged into classes and inheritance hierarchies established.

The Designer's Schema

A somewhat idealized picture of the designer's schema is shown in fig.5.



Aggregation: type design = designer, name, date, project



Generalization: design generalizes "layout", circuit, system

Figure 4. The two semantic datasbstractions

It contains hierarchies, a multiview specialization, versions and (multidesigner) locks. Introducing versions in a hierarchical environment presents special problems. In fig. 5 a version is modelled as an attribute of "cell". It is not a priori obvious what part of the hierarchical context of a cell should migrate with it from version to version. It is surely not advisable to change the status of the subcells when a new cell version is produced - we consistently take the subcells to be independent of their parents. But it is also not advisable to change the status of parentcells automatically, because that would mean that small modifications in a given cell may propagate throughout the system. A solution of the problem is proposed in [18] . Each cell has as an attribute a "version status". Only the status "actual" has the cell consistent upward and downward in the hierarchy. The past

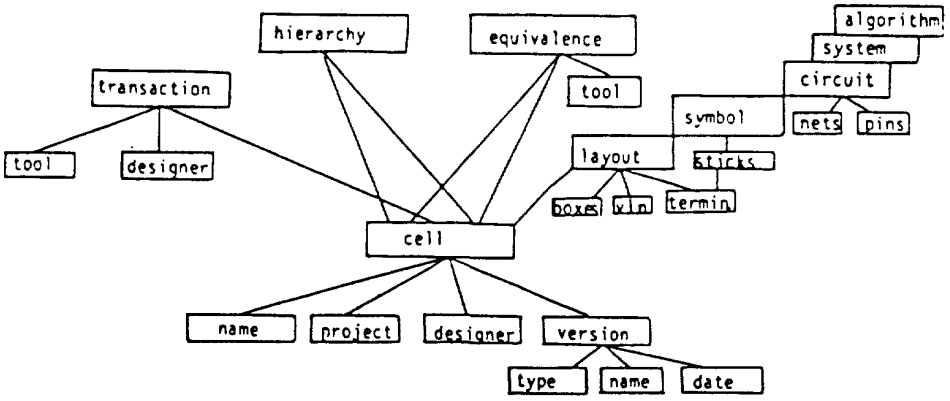


Figure 5. The Semantic Design Dataschema
The central object "cell" is a generalization of the objects in the views.
The hierarchy is modelled as an object which is an aggregation of design objects which possibly belong to different views."

history of the cell can be kept as "backup" for which neither the upward nor the downward hierarchy is necessarily consistent. Finally, when a designer checks out a cell for editing, its status changes to "working" and its subhierarchy can freely be used. It does not remain consistent with its upward hierarchy. A "working" cell can become actual when the "install" procedure is called. Only at that time shall the upward hierarchy be checked for consistency. This is done by flagging the effected changes upward and forcing installation of the parents recursively.

In addition to version control, the system should also provide for designer locks. Fig. 5 shows the appropriate part of the schema.

Database Implementation

The schema as presented so far consists of objects whose attributes have a fixed size. We have left the actual design data outside the modelling, thereby achieving a clear separation between *metadata* (information on the properties of the design data) and *design data*. This is necessary to keep the design management uncluttered of details which in any case only appropriate tools (editors e.g.) can handle - for the datamanager, the actual contents of a design is unimportant. The following will be required as properties of the implementation:

1. The actual (detailed) design data should be mapped to the system in as simple a way as possible, using its basic capabilities. A simple map to a file system may be sufficient.

2. The datamanager must be able to access the metadata with the highest possible speed, e.g. although multiuser the metadata must reside in virtual memory.

3. Design tools should be independent of the actual implementation of the database (see fig. 6).

These requirements can be satisfied by (1) defining a map of the design data onto the file system while respecting the schema, (2) by providing an "interface library" which translates the user's view (i.e. the schema) to the implementation and (3) by insuring that the interface functions handle the data in an efficient, i.e. properly localized, way. How to achieve this with not too much effort will be the subject of the next section.

4. Intelligent Definition of the Design Data

There is more to defining a design than the specification of the components of the design (even multiview). As we saw in the last chapter, some insight in the design is already captured by the "metadata", the description of the relations existing between design objects.

However, defining a design is an intellectual or intelligent process, in which one moves from a definition of one's insights at a higher level, verifies them interactively, refines them by defining objects at an immediate lower level and so on until one reaches descriptions that can be handled automatically by the available synthesis tools (and even after their use interaction may be warranted). The question arises whether a common denominator can be found between the various stages of development a design and his designer will go through, and which can be supported by the design system in an "intelligent" way.

Quite a few approaches have been proposed in the literature and we shall give an account of some of them at the end of this section. The approach we have found to be the most attractive is the so called "functional" approach. It is useful for analog [21] as well as for digital [22] design and yields some very promising results in both fields. Its main drawback seems to be that it uses a language that is a foreign to traditional programmers. It is however a natural language for expressing design actions.

In the functional approach we have adopted, any design step is seen to consist of two main components:
1. a transformation of "function" into "structure";

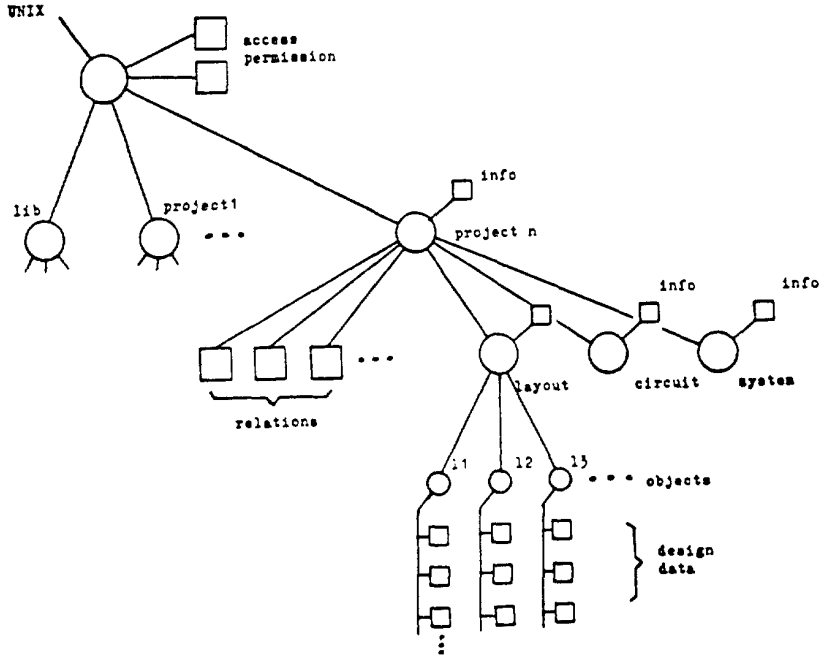


Figure 6. Mapping Design Data to the File System

2. a refinement of the types in which the function is expressed (with simultaneous refinement of the corresponding structure).
Any system that is to be designed - before the design is realized - is characterized informally by an "expected behaviour". Formally, this expected behaviour takes the form of a functional relationship between its inputs and outputs (i.e. at the end of all times, a certain set of input and output waveforms is consistent with the desired system). In many cases, the relationship may not be consciously known to the designer. Its only characteristic is that it exists: only the ports on which it has an influence are then part of the design. However, it will be advantageous if the designer has more precise knowledge of the expected behaviour, so that later on he will be able to verify it against the behaviour that actually has been realized. The mapping from function to structure shall in many cases be "intuitive" but can also be formalized if the environment in which the design takes place is formally defined, and refinement functions are available. For example, in the case of microprocessor design, the overall architecture of the system can be described in general terms in such a way that a direct mapping of a precise architectural description to

hardware via 'logic synthesis' and 'placement and routing' can be automated. Another, more ambitious, example is the automation of the design of dedicated array processors, where all the possibilities described above have to be used [23].
So far we have thought of the behaviour of a system-to-be-designed as a map between input and output waveforms, in which presumably the type of the inputs and outputs is known. Since a designer always wishes to handle his information at as high a level of abstraction as possible, he will choose high level data types between which his functional descriptions will be active (consciously or not). Refinement of these types hence become an important component of the design strategy. For example, at a given level of the design a signal may consist of a sequence of integers which later will turn out to consist of 16 bits. In one refinement, these 16 bits are presented in a bit serial fashion (the refinement of "integer" is now a sequence of 16 bits) and in another, the refinement will be 16 bits in parallel (in which case "integer" refines to "vector of 16 bits dependent on an event sequence"). Later in the design, the pins which carry the bits will turn out to actually carry electrical waveforms, and their type is further refined to "vector of 16 reals, dependent on continuous time").

Fig. 7 shows a representation of the design strategy described so far.

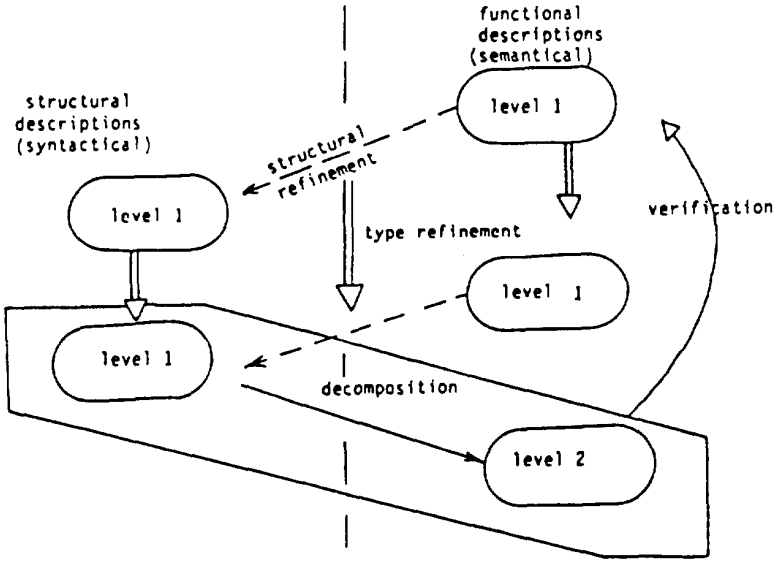


Figure 7. The design strategy with its two types of steps: conversion of function to structure and type refinement

In [21] and [13] examples of the use of the functional approach can be found.

In order to manage the design process described above, the data representation methods have to be further refined themselves. We are not yet ready for usage of a general purpose expert systems in the context of design definition - we find these systems still much too imprecise and too slow. Instead we shall take an intermediate approach where the basic AI representation method based on "frames" is followed, yet with a more rigid, application oriented schema. Central to our approach will be the need to handle the design data efficiently (and interactively). The designer should be presented with a logically uniform environment with which he can define his designs at any level of type refinement and abstraction. Many design operations, such as accessing specific fields in a design file (technically called the dereferencing of a pointer to a tuple), the defining and instantiating of design attributes, the maintainance of sets or sequences, are common to all levels and should be handled in a (level independent) transparant way. This is done by the concept of *abstract datatypes*. The dataschema presented in the previous section can usefully be extented to full-fledged *object-oriented data management*, because the latter supports the agregation and generalization needed with in addition the functions necessary to act on the data in a well controlled way.

Generalized Types

Much design data has the form of a collection (set or sequence) of *tuples*. E.g. a boxfile is a sequence of tuples consisting of coordinates of rectangles together with repetitions and mask information. The generic object tuple is characterized by a given set of operations that allow its definition (representation) and manipulation. Part of these functions are, of course, system provided. Aside from such general operations, specific tuples may have an additional collection of operations that are characteristic for them. E.g. if a tuple defines a lay-out primitive, it will be representable on a bitmap, and one will be able to edit it interactively. These additional operations will be defined in what we shall call - following the EC programming language which we use - a *cluster* [11] . A level higher than the tuples are agglomerates like "set" or "sequence". The system's cluster *set* will allow basic set definition and maintainance operations like adding or deleting elements, checking for the presence of an element and dereferencing. In the same way as with the basic tuples, one can define specialised functions specific for certain types of sets (e.g. a box file). Similarly, there will be *sequence*, *reference* and *one-of* clusters. An advanced object definition language allows forfor cluster definitions to be parametrized (grouped into classes) and to inherit properties from more general clusters.

Efficiency

Most datamanagement operations ultimately consist of accessing or storing objects. A major cause of inefficiency is the poor localization of the data describing the object. In standard general purpose programming languages like C or Pascal, the accessing of data is most often done by "dereferencing" a pointer to the data needed. Such an operation is extremely efficient because the data is typically already in the virtual memory of the computer and its localization has been properly taken care of by the compiler. When the data and metadata is in a (large) design database, it cannot be accessible in this way without cluttering the system and preventing multiuser access. The solution is to define a cluster *reference*. It will contain a dereferencing function (say ~) which will replace the standard dereferencing operator, and will in addition take care of the necessary data management: checking whether the required data is already in core (in which case the usual dereferencing is done), or fetching it when needed, update pointer counts, translate pointers to keys. All this is transparant to the application programmer in two ways: (1) he does not have to take care of the localization of the data in the system, (2) the basic operation is not dependent of the level at which he is programming. Efficiency results - when the functions in the cluster are properly programmed - because the database access function has now been made sensitive to the localization of the data, just as the program designer would have done it. It should be clear that the method explained here can be extended to higher datatypes so that more sophisticated "cache-like" handling of higher data aggregates becomes possible.

Layers of Implementations

Fig. 8 shows the layers of the system we finally have arrived at.
The intelligent users' semantic layer manipulates objects in the sense of procedural semantics. Only higher level representations of the design data and their relations are relevant. At this level, access to actual data is exclusively done with tools. The designer manipulates a functional or logic language like LISP or PROLOG which serves as database query language.

One level down is the object oriented datamanagement level where the efficient implementation of the design objects takes place. Characteristic for the approach described here is that the object definitions are precompiled. Here we use the already mentioned object extension of the C language - EC - to define the object types and their functions. Each type is a specialization of a cluster, and contains an aggregation of subtypes. In contrast to a general purpose object oriented system like Smalltalk, the possible schema is precompiled by the constructors of the system - this seems to be the price one has to pay to achieve efficiency. However, to keep this step also versatile, we use (1) interface

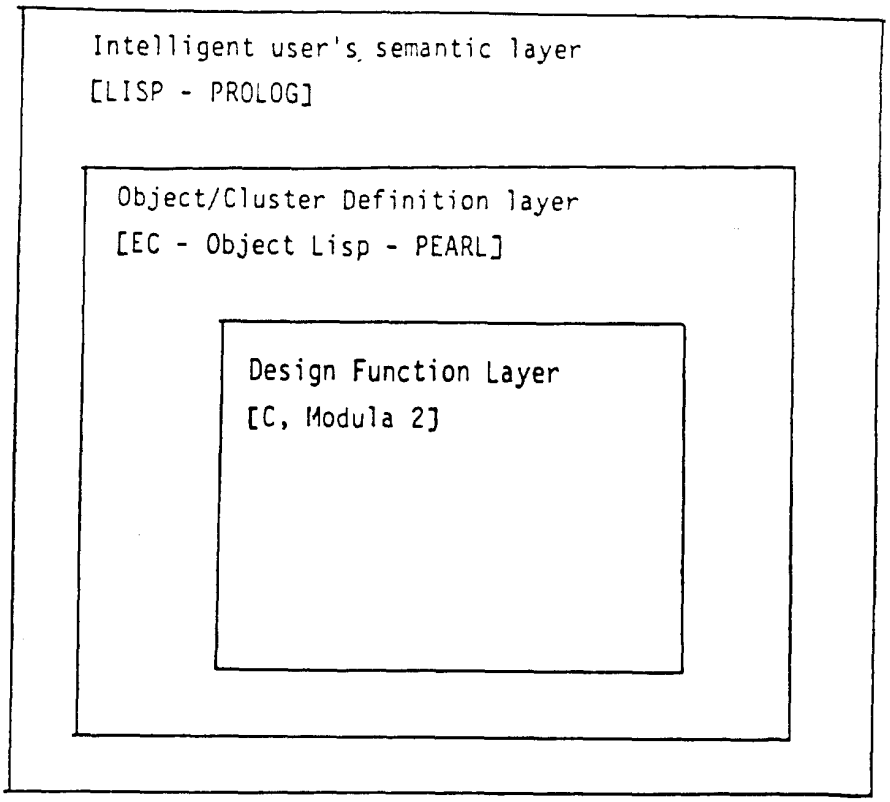


Figure 8. Layered Construction of an Efficient and Intelligent Datamanagement System

libraries that support users' views as explained in section 2, (2) general clusters and (3) an EC precompiler that allows the quick generation of EC code from some general properties.

A design system for the VLSI Array Processors has been build according to these principles, and is now available in prototype form [24] .

5. Discussion

A much used and misused word is "silicon compilation". It was originally intended to indicate the automatic assembly of parametrized pieces of layout. Later on a number of related design tools such as control generation, floor planning, global and local routing came to be designated with that name as well as cell generators. The term in its generality often overlooks the difficulties that constantly arrise with the actual intended and measured behaviour of circuits. A more germane characterization of the term is the one recently given by Gajski, Dutt and Pangrle in [25] in which the silicon compilation tasks are taken to consist of (a) the translation of functional descriptions into structural, (b) the instantiation of the lay-out of structural components, (c) their routing and placement, (d) the taking care of the integration in a system, and (e) the generation of test procedures.

The VLSI datamanagement methods presented here allow the constructors of a design system to integrate design definition and representation with verification in one consistent and open environment. A (delicate) balance can be struck between automation and interaction, whereby low level repetitions are relegated to the tools in the toolbox, general design intelligence becomes integrated in the design manager and where the superior intelligence of the human designer (especially in pattern recognition and electrical intuition) is needed, appropriate interaction both with tools and the manager is supported. In this way the dichotomy between "black box" silicon compilers and CAE workstation based systems is also resolved.

As a general concluding remark we can state that it is indeed possible to reconcile the stringent requirements of hierarchical/multiview design with efficient tools and intelligent users' interfaces. The solution we have presented uses an intermediate layer that consists

of an object oriented datamanager. Efficiency is obtained by somewhat restraining the designer's freedom to define new low-level datatypes (they are precompiled in the system). The object oriented definition style allows additionally for transparancy of the data to the tools, so that the system becomes truly portable as well.

6. Bibliography

References

1. J. McCarthy, P. Abrahams, D. Edwards, T. Hart, and M. Levin, *LISP 1.5 Programmer's Manual*, MIT Press, Cambridge, Mass ().
2. P. Roussel and R. Redheffer, "On the relation of Transmission Line Theory to Scattering and Transfer," *J.Math.Phys.* XLI pp. 1-41 Marseille-Luminy: Groupe d'Intelligence Artificielle, (1962).
3. R.H. Katz, "Managing the Chip Design Database," *Computer*, pp. 26-36 IEEE, (Dec. 1983).
4. D.W. Shipman, "The Functional Data Model and the Data Language DAPLEX," *ACM Transactions on Database Systems* 6(1) pp. 140-173 (March 1981).
5. P. Lyngbaek, *Information Modeling and Sharing in Highly Autonomous Database Systems*, Ph.D. Thesis, Univ. of So. California, Los Angeles (August 1984).
6. J.H. ter Bekke, *Database Design (in Dutch)*, Stenfert Kroese, Leiden (1983).
7. A. Goldberg and D. Robson, *SMALLTALK-80 - The Language and its Implementation*, Addison-Wesley Pub. Co., Reading, Mass. (1983).
8. M. Minsky, "A framework for representing knowledge - Chapter 6," in *The Psychology of Computer Vision*, McGraw-Hill (1975).
9. P.H. Winston, *Artificial Intelligence*, Addison-Wesley, Reading Mass (1984).
10. J. Katzenelson, "Introduction to Enhanced C (EC)," *Software Practice and Experience* 13 pp. 551-576 J. Wiley, (1983).
11. J. Katzenelson, *The Enhanced C Programming Language - Reference Manual*, Dept. of Electrical Engineering - Technion, Haifa (1985).
12. J. Backus, "Can Programming be Liberated from the Von Meumann Style? A Functional Style and its Algebra of Programs," *Comm. of the ACM* 21(8) pp. 613-641 (Aug. 1978).
13. J. Annevelink, "A Hierarchical Design System for VLSI Implementation of Signal Processing Algorithms," in *Proc. Intl. Conf. on Mathematical Theory of Networks and Systems - MTNS-85*, , Stockholm, Sweden (June 1985).
14. J. Annevelink, P. Dewilde, T.G.R. van Leuken, and J. Fokkema, "Hierarchical Verification of VLSI Artwork," pp. 44-52 in *Proc. IEEE Symp. on Circuits and Systems - ISCAS*, (1984).
15. T.G.R. van Leuken and J. Liedorp, "An Hierarchical Technology Independent Design Rule Checker," pp. 2.2-2.50 in *The Integrated Circuit Design Book*, ed. P. Dewilde, Delft University Press (1986).
16. J. Fokkema and T.G.R. van Leuken, "An Efficient Datastructure and Algorithm for VLSI Artwork Verification," in *Proc. IEEE ICCD Conf.*, , New York (1983).
17. J.L. Bentley and T.A. Ottman, " "Algorithms for Reporting and Counting Geometric Intersections"," *IEEE Trans. on Computers* 6-28(9) pp. 643-647 (Sept. 1979).
18. P. Dewilde, *The Integrated Circuit Design Book*, Delft University Press, Delft, the Netherlands (1986).

19. U. Lauther, "An O(N log N) Algorithm For Boolean Mask Operations," in *ACM IEEE 18th Design Automation Conference Proceedings*, , Nashville, Tennessee (June 1981).

20. J.M. Smith and D.C.P. Smith, "Database Abstraction: Aggregation and Generalization.," *ACM Trans. on Database Syst.* 2(2) pp. 105-133 (June 1977).

21. R.T. Boute, *System Semantic and Circuit Theory*, Dept. of Computer Science, Nijmegen, the Netherlands (1983).

22. P. Dewilde and A.C. de Graaf, "APPLY: an Applicative Network Description Method," in *Proc. ICCAD*, , Santa Clara (1984).

23. K. Jainandunsing and E.F.A. Deprettere, "Design and VLSI Implementation of a Concurrent Solver for N-Coupled Least-Squares Fitting Problems," *IEEE Journal on Selected Areas in Communications* 4(1)(Jan. 1986).

24. J. Annevelink and P. Dewilde, *Object Oriented Data Management for VLSI Design Based on Abstract Datatypes*, Techn. Rept. Network Theory Section, Dept. of Electrical Eng., Delft Univ. off Techn., Delft, the Netherlands (1986).

25. D.D. Gajski, N.D. Dutt, and B.M. Pangrle, "Silicon Compilation (Tutorial)," *Custom Integrated Circuit Conference*, pp. 102-110 (May 1986).