HIFI

A Design Method for Implementing Signal Processing Algorithms on VLSI Processor Arrays.

Jurgen Annevelink

HIFI

A Design Method for Implementing Signal Processing Algorithms on VLSI Processor Arrays.

HIFI

A Design Method for Implementing Signal Processing Algorithms on VLSI Processor Arrays.

Proefschrift

ter verkrijging van de graad van doctor aan de Technische Universiteit Delft, op gezag van de Rector Magnificus prof.dr. J.M. Dirken, in het openbaar te verdedigen ten overstaan van een commissie aangewezen door het College van Dekanen, op dinsdag 5 januari 1988, te 14.00 uur

door

Jurgen Annevelink

elektrotechnisch ingenieur

geboren te Laren (Gld) thans Lochem

Dit proefschrift is goedgekeurd door de promotor,

Prof.dr.ir. P. Dewilde

CONTENTS

7. *Voor het integreren van ontwerphulpmiddelen (tools) in een ontwerpomgeving zijn relationele database systemen niet toereikend wegens de beperkte abstractiemechanismen en het ontbreken van een versiemechanisme.*

8. *Het formuleren van een formeel model van het ontwerpproces is een belangrijke stap, welke kan dienen als basis voor het ontwikkelen van een ontwerpsysteem, wanneer het model de ontwerp-objecten en hun onderlinge relaties identificeert en klassificeert.*

9. *Een object-georienteerde taal heeft de volgende eigenschappen:*

   — *datatype abstractie*

   — *communicatie d.m.v. message passing*

   — *type inheritance*

10. *Hybride object-georienteerde talen, zoals Objective C, combineren een hoog abstractieniveau en grote modulariteit met een efficiente implementatie en vormen een uitstekende basis voor het ontwikkelen van een geintegreerd ontwerpsysteem.*

11. *Het leren programmeren behoort gericht te zijn op het leren toepassen van abstractie en (de-)compositie mechanismen met behulp van een hoog niveau programmeertaal, zoals LISP.*
    Abelson, Sussman and Sussman: 'Structure and Interpretation of Computer Programs', MIT press 1985

12. *Het euvel van 'herenaccoorden' zal verdwijnen als er meer vrouwen op hogere managementposities worden benoemd.*

Stellingen behorende bij het proefschrift:

HIFI

A Design Method for Implementing Signal Processing Algorithms on VLSI Processor Arrays.

door

J. Annevelink

1. *De vraag: 'Wat moeten wij doen ?', als grondslag van het ethisch handelen van een ingenieur kan niet worden losgemaakt van de vraag: 'Wat geloven wij ?'.*

   Greteke de Vries, 'Het ethisch denken van enkele Delftse technici', doctoraalscriptie, Faculteit der Godgeleerdheid, Rijks Universiteit Leiden, april 1987.

2. *Verdergaande internationalisering en concentratie van het bedrijfsleven vereist verdere ontwikkeling van internationale wetgeving gericht op het voorkomen van misbruik van monopolie- of oligopolieposities.*

3. *Het verdient aanbeveling met name de reisbudgetten van de onderzoeksgroepen aan de Technische Universiteiten te vergroten zodat men door het ontwikkelen van internationale contacten ook daadwerkelijk mee kan doen aan onderzoek van internationaal niveau.*

4. *Hoewel het jammer genoemd kan worden dat hoog opgeleide technici Nederland verlaten, zijn er ook vele positieve aspecten te onderscheiden, zoals het vergemakkelijken van de toegang tot het onderzoek in buitenlandse (industriele) onderzoekslaboratoria en het bevorderen van de internationale uitwisseling.*

5. *Het vakgebied der experimentele psychologie en de relaties tussen dit vakgebied en vele voor computer-aided design belangrijke aspecten, zoals cognitieve modellen en mens-computer interactie, worden onvoldoende onderkend.*

6. *Formaliseren van het ontwerpproces en de ontwikkeling van geintegreerde ontwerp-omgevingen leiden tot nog meer toepassingen van geintegreerde schakelingen dan thans al het geval is.*

Samenvatting

In dit proefschrift wordt een ontwerp-methode gedefinieerd, HIFI genaamd, welke het mogelijk maakt om op systematische wijze een grote klasse van signaal bewerkings algorithmen te implementeren op systolische en wavefront arrays.

Systolische en wavefront arrays zijn voorbeelden van array processor architecturen, welke bij uitstek kunnen worden geimplementeerd met behulp van VLSI technologie. Dit omdat beide soorten arrays zijn opgebouwd met behulp van een groot aantal gelijke processor elementen (PE's), die in een regelmatige structuur zijn gerangschikt. Tevens zijn de verbindingen tussen de PE's regelmatig en lokaal. Het belangrijkste verschil tussen beide soorten array processors is dat in een systolische array de PE's worden gesynchroniseerd met een globale klok, terwijl in een wavefront array de data-communicatie zorgt voor de synchronisatie van de verschillende PE's.

Het model dat ten grondslag ligt aan de definitie van de ontwerp-methode is een combinatie van een process georienteerd model en een applicatief, functie georienteerd model. Het resultaat is een model dat een hoge mate van abstractie combineert met krachtige decompositie mechanismen en dat zeer geschikt is voor het definieren van een hierarchische ontwerp-methode. Het model laat zowel een top-down als bottom-up ontwerp-stijl toe.

De HIFI ontwerp-methode zelf bestaat uit een opeenvolging van twee verschillende soorten ontwerp-stappen:

- *refinement* of verfijning, welke het mogelijk maakt om de decompositie van een functie te definieren door middel van een Dependence Graph.
- *partitionering*, waarmee de Dependence Graph wordt afgebeeld op een zogenaamde Signal Flow Graph, welke een efficientere implementatie is van de functie beschreven door de Dependence Graph.

De ontwerp-methode wordt geillustreerd door een aantal voorbeelden, respectievelijk een algorithme voor het oplossen van een stelsel lineaire vergelijkingen en het transitive closure algorithm. Ook wordt aandacht

geschonken aan de architectuur van een prototype ontwerp-systeem, geschikt voor implementatie van de HIFI methode. De design database vormt een belangrijke component van zo'n geintegreerd systeem, omdat hierin het ontwerp en alle verdere daarop betrekking hebbende informatie zijn opgeborgen.

In het laatste hoofdstuk wordt de HIFI methode nader bekeken en formuleren we een aantal eisen welke gesteld moeten worden aan de software omgeving waarin een systeem als het hier voorgestelde kan worden ontwikkeld en gebruikt.

Het proefschrift bevat verder een drietal appendices in de vorm van (gepubliceerde) artikelen. Appendix A bevat een artikel, waarin een overzicht wordt gegeven van systolische en wavefront array processors en algorithmes. Het artikel in Appendix B beschrijft een procedure waarmee een algorithme, beschreven door middel van een Signal Flow Graph, getransformeerd kan worden naar een vorm welke direct afbeeldbaar is op een systolische array. Het artikel in Appendix C definieert een methode om op een systematische manier de ontwerp-data af te beelden en op te bergen in een ontwerp-database.

Summary

In this dissertation we define a design method, called HIFI, that makes it possible to systematically implement a large class of signal processing algorithms on systolic and wavefront arrays.

Systolic and wavefront arrays are examples of processor architectures, that are very much suited to VLSI implementation, because both systolic and wavefront arrays are build using a large number of similar processor-elements (PE's). The PE's are arranged in the form of a regular grid, while the interconnections between the PE's are regular and local. The most important difference between the two architectures concerns the synchronization of the PE's. In a systolic array the PE's are synchronized by a global clock; in a wavefront array the synchronization is achieved by adopting an asynchronous hand-shaking protocol for the communication between the processor elements.

The model underlying the definition of the design method is a combination of a process oriented model and an applicative, function oriented model. The result is a model that combines a high level of abstraction with powerful decomposition mechanisms. The model is used to define the HIFI design method, which allows both top-down and bottom-up design styles.

The HIFI design-method is based on two different design steps:

- *refinement*, which makes it possible to define the decomposition of a function with a Dependence Graph
- *partitioning*, used to project a Dependence Graph on a so-called Signal Flow Graph, which allows a more efficient implementation, of the function defined by the Dependence Graph.

The design method is illustrated by a number of examples, respectively an algorithm for the solution of a system of linear equations, and the transitive closure algorithm. In addition, we discuss the implementation of a prototype of the HIFI design system. An important component of an integrated design system will be the design database, which contains all the information relevant to a particular design.

In the last chapter we review the HIFI method with respect to the its computational model. We also discuss some requirements for the software environment in which a design system such as the one proposed here, can be implemented.

This dissertation has three appendices, containing previously published papers. Appendix A is a reprint of a paper, reviewing the influence of the basic VLSI device technology on VLSI processor architectures. The paper reprinted in Appendix B defines a procedure to transform a Signal Flow Graph (SFG) description of an algorithm in a similar SFG, that can be directly mapped onto a systolic array architecture. The paper in Appendix C defines a systematic method for interfacing design tools with a design database.

1. Introduction

The development of the Integrated Circuit (IC) technology, reflected in the increase in integration density and size of an integrated circuit, makes it possible to design and implement more and more complex systems. Whereas about 25 years ago the number of components on an integrated circuit or chip ranged from 10 to 100, state of the art, so-called VLSI chips may presently contain upwards of one million components, mostly transistors. This trend drives current design methods to their edge. The problems that become visible are especially related with:

- management of design complexity
- verification of the design

In this thesis we will describe a design methodology that should tackle the above problems for an important class of algorithms, namely those that have a regular and local data-flow. These types of algorithms are especially suited to VLSI implementation for a number of reasons. First, because of the regularity, the design complexity is reduced. By designing a suitable processing element and then duplicating it, one can form an array (1D or 2D) of processing elements. Second, the local data-flow is favorable, because in a VLSI chip the cost of communication, reflected by the length of the connections between different processor elements, quickly becomes the main bottleneck.

Array structured processor systems have long been the subject of active research as illustrated by the work of Von Neumann and others on cellular automata [Beck80]. The development of VLSI technology led to the definition and design of systolic architectures [Kung79]. A systolic array can be informally described as an array of pipelined processor elements operating in unison on a set of data. Systolic arrays may be designed to match the I/O and throughput requirements of many applications, both numeric and non-numeric [Kung87]. Numeric applications are found in the area of signal processing, e.g. filtering and radar signal processing and matrix arithmetic. Non-numeric applications include graph algorithms, e.g.

transitive closure and dynamic programming.

In order to be implemented on systolic arrays an algorithm must be specifiable in the form of a set of recurrences. Rao [Rao85] showed that a systolic array defines a set of recurrences and correspondingly that certain sets of recurrences, the so-called Regular Iterative Algorithms, are implementable on systolic arrays. The definition of these recurrences is only one part of a design however. The other aspect is the scheduling of the operations and the assignment of operations to processors, given the constraints imposed by a particular hardware implementation technology, or an existing multiprocessor architecture with its fixed pattern of interconnections. A mathematical framework for deriving a schedule and assigning operations to processors, for a restricted class of algorithms, the so-called RIA or regular iterative algorithms, was given by Rao [Rao85]. This work (re)established the prominence of the so-called dependence graph, as a basis for mapping a particular algorithm onto a regular multi-processor network, such as the systolic and wavefront type arrays described in [Kung84]. Complementing the work of Rao, Moldovan [Mold86] and Deprettere and Nelis [Neli86] showed how to compute a schedule and assign operations to processors given a limited number of processors.

In this thesis we will concentrate on the description of a design method that supports the design and implementation of algorithms with regular and local data-flow. The design method will be embedded in a prototype CAD system that provides a framework for incorporating a large variety of tools that can assist a designer in the task of defining a processor architecture as well as deriving the schedules, control flow and/or hardware structure of the processor elements.

The architecture of the design system is schematically shown in Figure 1.1. It reveals that the design system consists of three important parts, respectively (1) the database and associated Database Management System (DBMS), (2) the design tools and (3) the user interface and input/output devices.

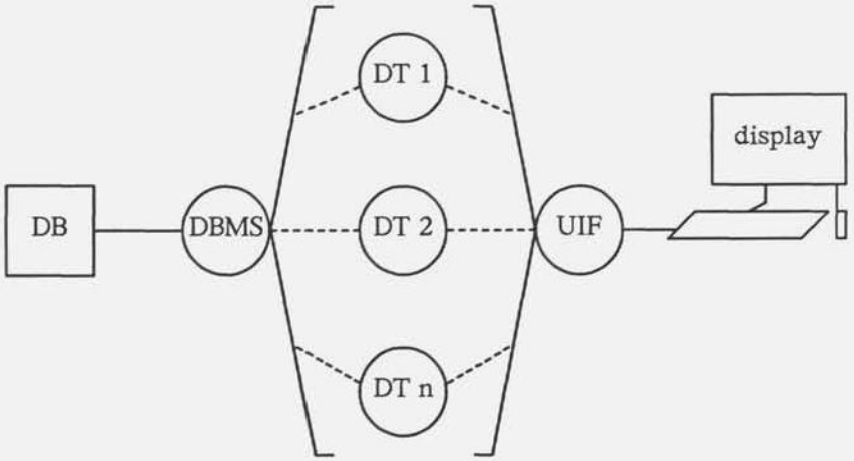

Figure 1.1. Architecture of the Design System

The database is intended to provide persistent storage for the design objects, that can be manipulated via the DBMS. The functionality of the DBMS is in large part determined by the data model on which it is based. The data-model defines the methods needed to define a data-schema, i.e. a precise definition of the types of the objects stored in the database, their relationships and the operations allowed on them. The best known data-model is the relational model [Date81]. The relational model allows data-schemas to be defined in terms of relations. A relation defines types comparable to the tuple or record types found in programming languages. The relational model is attractive because of its simplicity and its simple implementation; a relation can be stored as a table, which can be efficiently searched. A disadvantage is the low level of expressivity. Because the design tools have to construct their internal data-structures from many pieces of low-level information, that are to be retrieved separately from the database, the interfacing of the design tools with the database is complicated. This is not only inefficient, it also impairs the data-independence, since the design tools need to incorporate detailed knowledge regarding the mapping of their internal data-structures onto the relational types provided by the database.

The above considerations (amongst others) stimulated the development of richer, so-called semantic data-models that provide a richer set of data-types, to facilitate a more direct mapping from the data-structures used internally by the design tools to database types. The further development of these data-models is an active area of research, which is vital also for the development of design systems. The demands posed on a DBMS by a design system differ also from more conventional transaction oriented systems, in that the database is required to keep track of the progress of a design, reflected in the successive versions and alternative solutions produced by the designer during the evolution of a design.

The design tools are intended to provide the complex and highly specialized operations that can not be captured by the data-model of the DBMS. They operate on the data stored in the database, modify it and then store the resulting data back in the database. The input from the designer, commands etc., is obtained from the user interface, which interfaces the design tools to the input/output devices. Input/output devices usually consist of a high-resolution graphics display, a keyboard and a mouse.

A critical aspect is the mechanism provided to control the design tools and to represent various aspects of the design data to the designer. The flexibility of the design system is increased greatly, if it is possible to control various design tools, operating concurrently and accessing the same database. This possibility is offered by various technologies for implementing user-interfaces, e.g. the X-window management system and specialized class libraries offered by various object oriented programming environments, such as SMALLTALK [Gold84]. An important difference between these technologies is whether design tools are assumed to be independent processes, managed by the operating system of the host computer, under the control of a window manager process, such as in X, or that the design tools are merely procedures run under control of a controller, such as is the case in the SMALLTALK environment. In the latter case, the granularity of the design operations can be much made much smaller.

There are also many trade-offs to be made regarding the implementation of the DBMS. On the one hand the design tools may include all data access methods themselves and thus be completely responsible for maintaining the consistency of the database. Such a solution however requires the database to be very simple, otherwise the overhead of including the DBMS functionality in all design tools becomes prohibitive. On the other hand, it is possible to encapsulate the functionality of the DBMS in a separate process, controlled by the operating system. Given current networking facilities this also facilitates a distributed environment in which the database is (physically) located at a specialized database server, that may include special hardware facilities to increase the efficiency of the DBMS.

1.1 Management of Design Complexity

There are a large number of aspects involved in managing the complexity of a design. First and forall there is the need for a proper decomposition of the design. Typical for the design of VLSI chips is the use of an hierarchical and multi-level design method. The ICD system [Dewi86] for example distinguishes between a number of levels or views of a design. Each of these views will contain a specification of the design as an hierarchical composition of modules or cells and submodules or subcells. There are no à priori relations between the cells in one view hierarchy with those in another, although such relations may exist. In fact, they will exist and are created and/or used by synthesis and verification tools. For example, a circuit extractor will generate a network description that can be mapped one to one onto the layout description of the cell to which it is applied. The extracted network description can then be compared to another network, so as to verify whether the two network descriptions are equivalent.

Hierarchical and multilevel design is one way of reducing the design complexity. Another, equally important method is that of stepwise refinement. Stepwise refinement is a basic technique used e.g. in software design. It is used to separate the design of the interface or abstraction of a module (e.g. a procedure or function), from its implementation. By first designing the interface it becomes possible to test/verify in an early stage of development the interaction of a module with its environment. Once the

interface is tested/verified, one has to verify the implementation only with respect to the specification of the interface. Stepwise refinement works only when it is considerably easier to specify the interface of a software or hardware module, than it is to verify its implementation.

A fair amount of complexity is added to a design system, when it is required to keep track of the evolution of a design. Slightly different requirements, e.g. with respect to speed, will usually require only a few modifications to a few modules. In such a case it is appropriate to view the two designs as versions or alternatives of a single design.

Another factor that influences the complexity of a design is the implementation technology. It makes a big difference whether a chip must be designed using some gate array technology, standard cells, or whether a full custom implementation is required. Similarly, for software, the language and other design tools, e.g. compilers and debuggers, used to implement modules may significantly effect the difficulty of doing so. In the case of algorithmic design, which we consider in this thesis, it is likewise the combination of the design methodology and the design tools that support it, that determine the ultimate complexity of a design.

1.2 Design Verification

The importance of design verification goes without question. The costs involved in fabricating a prototype of a VLSI chip and the time needed to do so, are high. It is therefore important to verify the correctness of a chip *before actually fabricating the prototype, to maximize the chance that the* prototype functions correctly and to reduce design costs. Current verification methods are mostly based on simulation techniques. To an executable specification of the chip, or a part thereof, a series of inputs is fed. The values on the appropriate outputs have to be compared with the expected results, e.g. by comparing the results with the results of simulations in a different view of the design. The input sequences are chosen to maximize the chance of detecting design errors at the outputs.

There are many problems associated with this type of verification. First of all, the design of a set of simulation inputs is very difficult. The fault-

coverage, a measure for the quality of the simulation inputs, is difficult to determine in general. An alternative approach is to prove the correctness of the design. This requires the semantics of the specification/design language to be precisely specified. Currently there is a lot of research being done on the semantics of programming languages. This research has applicability to hardware design as well.

For the design methodology described in this thesis the correctness will be guaranteed mostly by construction. Given a correct transformation and the system will allow only correct transformations, the result will be correct also. In cases where that is difficult to achieve, descriptions will be executable, so that a design can be simulated.

1.3 Overview

In chapter two of this thesis we will review the area of array processors and systolic arrays in particular. Two papers added as appendices A and B provide respectively a more detailed review of array processors and array processing, including VLSI implementation and technology trade-offs [Kung83] and an algorithm for systolizing systems described by a Signal Flow Graph [Kung84a].

Chapter three continues with a review of a number of models and languages used for describing concurrent systems. The models are compared on a number of criteria that are on the one hand derived from an analysis of the design process and on the other hand related with the special requirements posed by VLSI implementation, e.g. locality of interconnections and regularity.

Chapter four gives a definition of the design method which we propose and have called HIFI. The computational model underlying the design method forms the main part of chapter 4.

Chapter five discusses the two main design steps, respectively function refinement and function implementation, in more detail.

A prototype system is discussed in chapter six. There we also discuss the database requirements of the HIFI system. A systematic data-management

strategy, based on the definition of the data-structures used by the design tools, is described in a paper [Anne88], added as appendix C.

Chapter seven discusses two examples. The first is a system for implementing the transitive closure algorithm. The second is the design of a system implementing an algorithm for the least-square solution of a system of linear equations.

Finally the results of the research described in this thesis are discussed in chapter eight.

2. VLSI Array Processors

Influenced by the rapid progress in VLSI device technology many algorithms have been developed that can be implemented on so-called *Systolic Arrays* [Kung79]. A Systolic Array is a regularly interconnected set of identical processing elements arranged in the form of a grid. The qualification 'systolic' derives from the fact that in a systolic array all processor elements perform their operations rhytmically on the beat of a global clock. A systolic array is thus per definition a synchronous system.

An important generalization of systolic arrays are the so-called *Wavefront Arrays* [Kung82, Gal-82]. The name 'wavefront' array is derived from the fact that the propagation of the computational activity on such an array resembles the propagation of a wave. The difference between systolic and wavefront arrays is the synchronization of the processor elements. In a wavefront array, all processor elements have their own local clock and the communication between processor elements is by means of a handshaking protocol, i.e. processors can wait for one another if data is not available. The asynchronous communication allows processor elements to be locally synchronized, removing the problems associated with synchronizing a large number of processor elements. The disadvantages of wavefront arrays are mostly the increased complexity of the processor elements and the possibility of deadlock, i.e. the situation in which two or more processors are waiting for each other to produce or consume data. A more extensive discussion of wavefront arrays, including a prototype design for a processor element, can be found in Appendix A.

2.1 Design/Implementation of VLSI Array Processors

Probably the best known example of a systolic array is the systolic array for (banded) matrix-matrix multiplication of Kung and Leiserson [Mead80]. The array consists of a set of hexagonally connected processor elements, so-called *Inner Product Step* processors, that consist of a multiplier, an adder and a number of registers to buffer the input data, while a computation takes place. It is easy to see that the matrix product $C = (c_{ij})$ of $A = (a_{ij})$

and $B = (b_{ij})$ can be computed by the following recursion:

$$\begin{aligned} &\text{for i, j, k from 1 to n do} \\ &\quad c_{ij}^{1}=0 \\ &c_{ij}^{(k+1)}=c_{ij}^{(k)}+a_{ik}b_{kj} \\ &\quad c_{ij}=c_{ij}^{(n+1)} \end{aligned} \tag{2.1}$$

If A and B are $(n \times n)$ band matrices of band-width w_1 and w_2 respectively, then the above recursion can be evaluated by pipelining the a_{ij}, b_{ij} and c_{ij} through the array of hex-connected *Inner Product Step* processors shown in Figure 2.1, for the case $w_1=w_2=4$.
In order to verify that the array in Figure 2.1 indeed implements the recursion (2.1), the data-flow in the array has to be studied in detail. One approach is to make a series of successive snapshots to follow the flow of data in the array and to verify that the successive $c_{ij}^{(k)}$ indeed accumulate the partial sums $a_{ik}b_{kj}$. Chen [Chen83] formalizes this approach and shows that the computation implemented by the array can be found by solving a set of space-time recursion equations for the least fixed point.

Experience shows that it is usually possible to design several different systolic arrays in order to implement a particular set of recursions. It is therefore natural to ask whether we can systematically generate all systolic arrays that correctly implement a particular recursion. The problem is reversed in the sense that we are looking now for a methodology to synthesize a systolic array starting from the recursion equation that defines its behavior.

The definition and implementation of systolic arrays can be simplified if we use a model for their definition that abstracts from the actual timing and synchronization of the processor elements. S.Y. Kung [Kung84] uses *Signal Flow Graphs* for this purpose. The SFG's used by Kung represent computations by nodes that are connected by directed edges that have a weight representing the number of data values (initially) present on the edge. The nodes operate by taking the first data-value from their input edges to compute values that are appended to the output edges, one value to

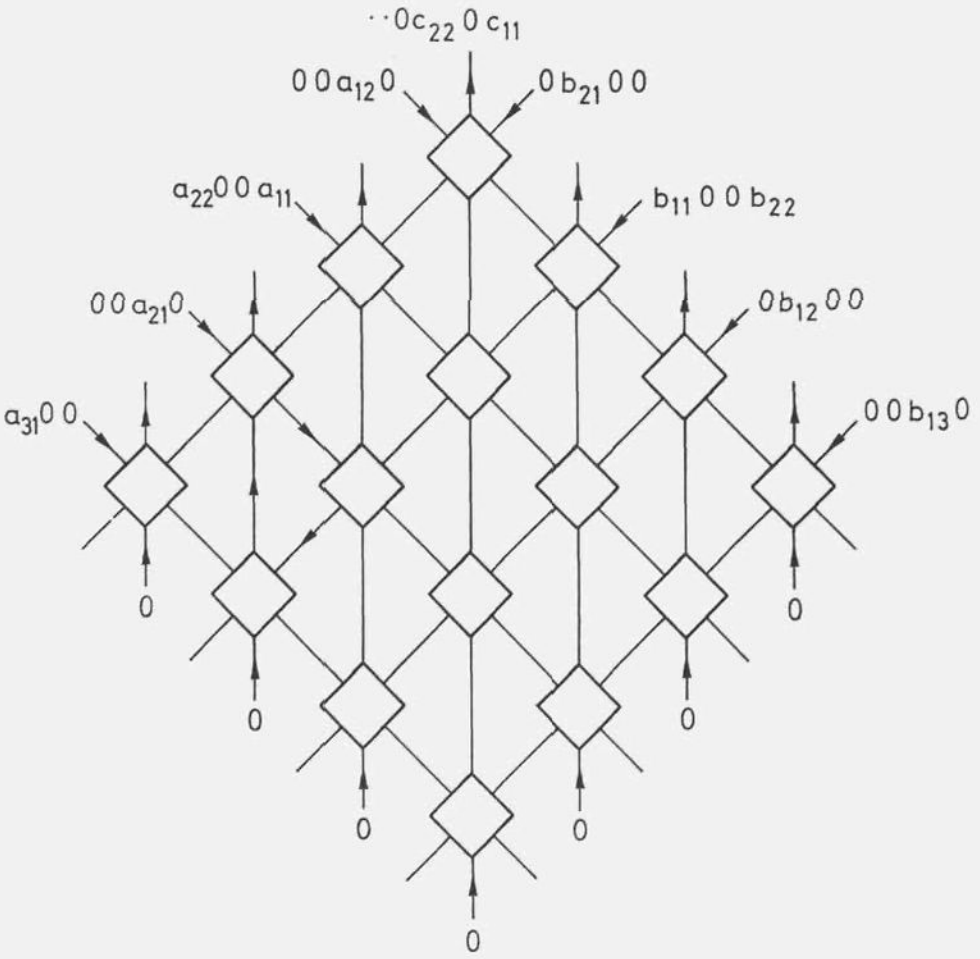

Figure 2.1. Systolic Array for (banded) matrix-matrix multiplication

each edge. In fact, the SFG's used by Kung in [Kung84], are very similar to the Data Flow Graphs (DFG) used by other researchers. This is also recognized by Kung, who shows that a SFG can be easily transformed into a DFG.

The recursion (2.1) is implemented by the SFG shown in Figure 2.2. The SFG is much easier to interpret and verify then the corresponding systolic

array in Figure 2.1, due to the fact that the k-indices of the computations in a shapshot are all equal. All that is required by the SFG representation is that nodes are not dependent on one another for their input data. This condition is easily met if we require that the SFG doesn't contain zero weight loops. Note also that the data entered in the SFG is not interleaved with zero's as is the case for the systolic array.

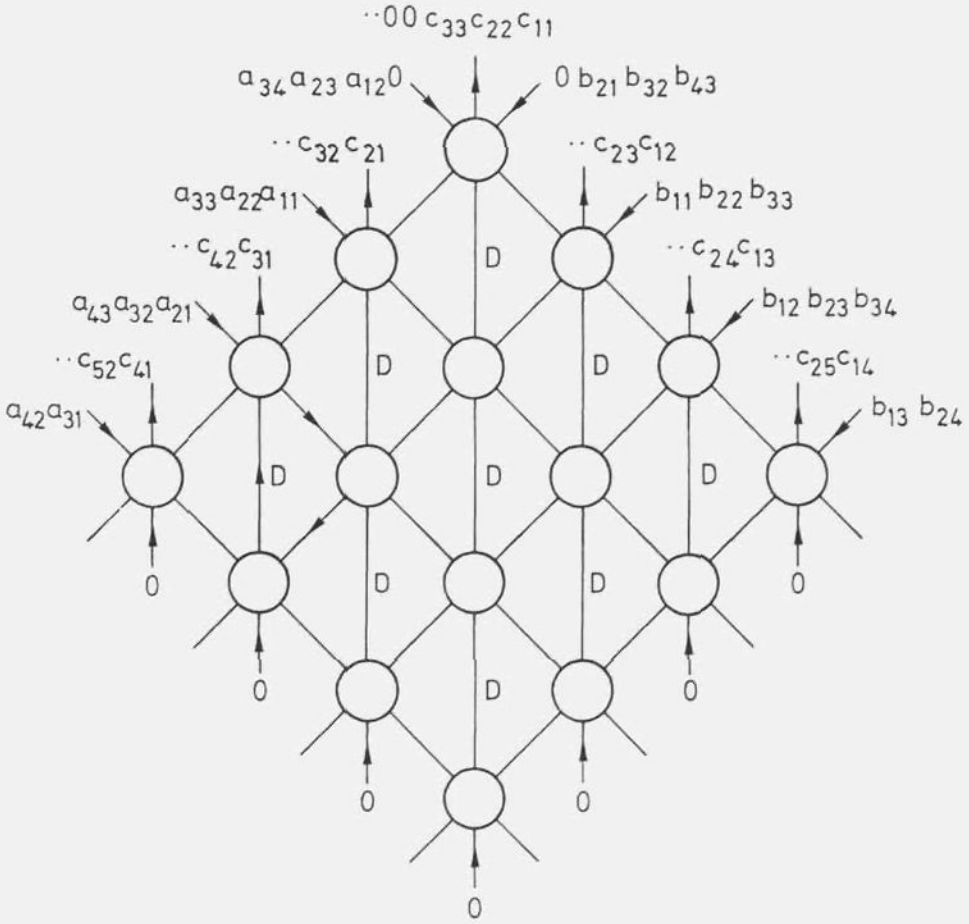

Figure 2.2. SFG for (banded) matrix-matrix multiplication

A SFG as shown in Figure 2.2 can be 'automatically' transformed to a systolic array using the procedure defined in Appendix B. It is only required that the SFG is computable. The procedure described in Appendix B starts by *temporally localizing* the SFG, i.e. the SFG is transformed into a computationally equivalent one such that the weight of every edge is ≥ 1. The procedure is based on two simple rules; (1) *time-rescaling*, which is used to rescale the time delays by a positive factor α, in order to localize loops and (2) *delay-transfer*, which is used to distribute the delays evenly over the edges of the SFG. A temporally localized SFG can be transformed into a systolic array by combining a delay from all inputs of a node with the node itself, in order to form a basic systolic processor. The result of temporally localizing and systolizing the SFG of Figure 2.2 is the array of Figure 2.1. The delay rescaling that was necessary equaled $\alpha = 3$; this is the minimum rescaling, since the SFG contains a loop with three edges.

In order to improve the efficiency of systolic array with $\alpha > 1$, a group of α consecutive processor elements may share a single arithmetic unit, without compromising the throughput rate as shown by Kung [Kung84].

Although the SFG representation offers many advantages over the representation shown in Figure 2.1, there is still a choice regarding the SFG used for implementing a set of recursions. A SFG forces a certain order of evaluation, i.e a schedule, on the computations specified by the recursion. This can be modeled by representing the recursion as a *Dependence Graph* (DG). Different SFG's can be found by projecting the DG in different directions on a lower dimensional SFG. In order to conform to the local interconnection constraint posed by a VLSI implementation, the recursion has to be rewritten in a so-called single assignment, *localized* form, meaning that variables may occur only once on the left hand side of an equation and that dependencies between the variables on the left and right hand-size of the recursion equations have to be constant and independent of the value of the indices. The DG representing the recursion is found by mapping the variables occurring on the left hand side of an equation on the grid points of an index space; the dependencies between these variables are represented by directed arcs between the corresponding grid points. The extent of the index

space can be defined by a set of constraints that define the points contained in it. The definition of these index spaces and their mapping on systolic arrays are discussed in [Rao85].

The design method discussed in this thesis (cf. chapter four) also uses Dependence Graphs to specify the recursions. Contrary however with the approach taken by Rao, we will assume that the DG is defined by a sequence of successive decomposition or refinement steps. The partitioning step can similarly be decomposed. It is also possible to consider a sequence of refinements and construct a DG by substituting the DG's in one another. As a result, in order to find a suitable model for our DG's and SFG's, a major point of attention in our research have been languages for describing and defining concurrent systems. This will be described in more detail in chapter three.

2.2 Other Aspects of VLSI Array Processor Design

In general there are a large number of aspects that influence the design and implementation of systolic and wavefront arrays. In addition to the considerations mentioned above, regarding the definition of the recursions and the associated DG's, there are also considerations regarding:

General purpose vs. special purpose processing elements

The development costs associated with a systolic/wavefront array are such that it is mandatory to have as large an application area as possible. It may be worthwhile to have programmable processor elements and/or flexible interconnections so as to increase the number of algorithms that can be implemented on a particular array.

Granularity of operations

The basic operation performed by each processor element may range from a simple bit-wise operation through word-level multiplication and addition to execution of complete programs. The level of granularity is determined by the choice of processing elements which will depend mostly on technological and implementation constraints, e.g. I/O limitations and throughput

requirements.

Partitioning

In general, when problems of arbitrary size have to be processed on an array of a fixed size the problem must be partitioned so that the large problem may be efficiently solved on the fixed size array. Several approaches are possible. One approach operates by partitioning the DG of the algorithm such that the individual partitions can be mapped onto the processor array. The global control necessary to ensure correct sequencing of the algorithm partitions as well as storage of intermediate data has to be added to the description of the array. This so-called Local Sequential Global Parallel approach [Jain86], increases the amount of memory required externally. Another approach, called Local Sequential Global Parallel [Jain86], operates by clustering neighbouring nodes in the DG and mapping them on a single processor. This requires additional control regarding the sequencing of the operations as well as local storage of values to be added to the description of the processor elements. Yet another approach is to restate the algorithm, such that it becomes a collection of smaller problems that are similar to the original problem, but can be solved by the given systolic array.

Fault Tolerance

For large arrays the inclusion of a certain degree of fault tolerance has to be considered, since the reliability of the processor array degrades rapidly when the number of processors increases.

Synchronization

An important issue for systolic arrays is the synchronization of the processor elements. Depending on the size of the processor array and the layout of the clock-distribution network, the skew introduced by the fact that clock-lines differ in length, will degrade the performance, since it lowers the maximum allowable clock-frequency. An alternative to the design of a globally synchronous array is to replace the global synchronization by self-timed data-driven synchronization by means of an

asynchronous hand-shaking mechanism, as in a wavefront array. The disadvantage of this is that the handshaking mechanism adds overhead to the communication between processor elements. This overhead can only be justified if the operations implemented by a processor element are of a sufficiently high degree of granularity.

Integration in existing systems

The problems associated with integrating array processor systems such as systolic arrays into existing computing networks may be non-trivial because of the high I/O bandwidth required by the array processor. If a host-processor can't keep up with the processing rate of a systolic array this may require insertion of special memory buffers, or even a hierarchy of successively faster memories. For some applications however, the array processing system actually reduces the I/O requirements that would otherwise be put on the host processor, e.g. in radar signal processing (adaptive beamforming), or image processing (feature extraction, image enhancement). Naturally these are ideal applications for systolic processors.

In this thesis, we will be concerned mostly with the definition of the computations performed by a systolic/wavefront array, as discussed in section 2.1. In the next chapter we will therefore study a number of models and languages for specifying concurrent systems.

3. Models and Languages for Concurrent Systems

In this chapter we will give an overview of a number of languages/models for specification and design of software and hardware systems. We will first discuss a generic model of the design process that is applicable to both software and hardware design. The differences between software and hardware design become visible only at more detailed levels, where technological constraints have to be taken into account. Next we will give an overview of a number of languages and models that have been developed for the description and design of concurrent systems. This overview will outline two trends, process oriented modeling and applicative languages. Both have been developed in efforts to reduce the complexity associated with modeling and verifying the correctness of large (software) systems. In addition we will describe a number of languages that combine ideas from these areas. The last section of this chapter contains a discussion and comparative evaluation of these languages and models and identifies desirable properties for a design model/language.

3.1 The Design Process

A popular view of the design process is to partition it in two phases. Specification is separated from implementation and verification. In this view one first specifies a system completely in a formal language at a high level of abstraction. Then the implementation issues are considered and a program or system design are developed and verified with respect to the specification. The above simple view of the design process can not be maintained in light of design methodologies such as stepwise refinement and object oriented programming [Gold83], that have been developed over the past 20 years. It became obvious that the partitioning between specification and implementation is entirely arbitrary. Every specification is an implementation of some other higher level specification (cf. figure 3.1).

The standard software development model holds that each step of the development process should be a "valid" realization of the specification. By "valid" we mean that the behavior specified by the implementation is equal

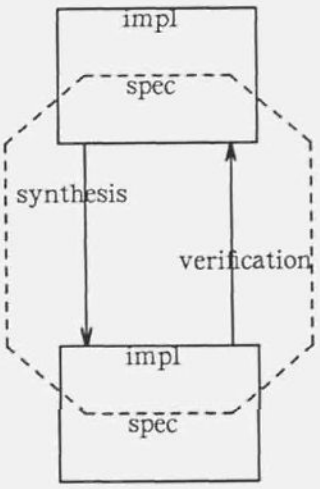

Figure 3.1. View of the Design Process: Successive Specification and Implementation Steps

to that defined by the specification. This equality has to be verified. In practice one finds that many design steps violate this validity relationship between a specification and its implementation. Rather than providing an implementation of the specification, they knowingly redefine the specification itself. Implementation is a multiple-step process and many of these steps are not mathematically valid, i.e. they don't implement the specification, they alter it.

There are two important reasons for specification modifications: physical limitation and imperfect foresight. The systems we design are build from components that have limitations, such as speed, size and reliability. Often it will be possible to find a cost-effective partial solution, rather than a total solution. This introduces either a restriction that limits the domain of input or introduces the possibility of error. In the latter case it is necessary to define what to do when an error occurs. In either case the semantics of the specification has been changed due to an implementation decision. The second source of specification modification is our lack of foresight. The systems we specify and build are complex. It is virtually impossible to

foresee all the interactions in such systems. During implementation these implications and interactions are examined in more detail. Often we find undesirable effects or incomplete descriptions. This insight provides the basis for refining the specification appropriately. The place where the design modification is inserted depends upon the implementation decisions that are affected.

It follows that the design process is not a simple two step process, specification and implementation, but that the design process consists of a sequence of specification and implementation steps, where the implementation at one level serves as the specification at the level below. Interleaving of specification and implementation steps is further complicated by the fact that certain implementation choices may actually change the (semantics of the) specification above (cf. Figure 3.1).

The interleaving of specification and implementation is due to the fact that at any one level of design one wants to limit the amount of complexity or detail that must be considered. Two common and effective approaches to accomplish this are *decomposition* and *abstraction*. By decomposing a design task into subtasks, the complexity of the design is effectively reduced to that of designing and combining the individual subtasks, because the subtasks can be treated independently. For many problems however, the smallest separable subtasks are still to complex to be designed in a single step. The complexity of such tasks must be reduced via abstraction. Abstraction provides a mechanism for separating those attributes that are relevant in a given context from those that are not, thus reducing the amount of detail that one needs to come to grip with at any one time.

Decomposition and abstraction techniques can be identified in conventional approaches to IC design. The so-called multi-level hierarchical design method employs abstraction by the introduction of multiple levels, such as algorithmic, register-transfer, logic gate, switch-level etc. A design will usually be described at a number of these levels. At each level the designer can then decompose the design to reduce the remaining complexity. Usually, a cell or module will be composed of a number of subcells or submodules, which in turn are composed of subcells etc. The cells are

related hierarchically.

Such an approach may be effectively captured by a so-called Y-chart [Gajs83]. The Y-chart shown in Figure 3.2 is a convenient and succint description of the different phases of designing VLSI systems. The axes correspond to the orthogonal forms of system representation. The arrows represent design procedures that translate one representation into another. While many different design approaches and their corresponding Y-charts are possible, design is typically carried out through a process of successive refinements. In this process a components functional specification is translated first into a structural description and then into a geometrical description in terms of smaller subcomponents; the functional descriptions of each of these subcomponents must be translated into structural and geometrical descriptions of even smaller parts and so on.

The principal weakness of this approach lies in the diversity of models and associated notations used to describe a design at the various levels. Attempts to "unify" the different levels are mostly based on imposing the same decomposition at all levels of description. By thus fixing the "structure", one can view the levels as different aspects of a cell or module, e.g. its behavior or its topology.

Besides the difficulties associated with imposing a uniform decomposition at all abstraction levels, this approach does, in my view, not solve the principal difficulty, which is the wide range of underlying models. This makes it very difficult to devise a formal method for verifying that a description of a cell at one level of abstraction indeed represents the cell as described at another level.

The best known aid to abstraction used in programming is the self-contained, arbitrarily abstract, function, by means of an unprescribed algorithm. A function, at the level where it is invoked, separates the relevant details of "what" from the irrelevant details of "how". In addition, by nesting functions, one can easily develop a hierarchy of abstractions. The nature of abstractions that can be achieved through functions is limited however. Functions allow us to abstract single events, the application of

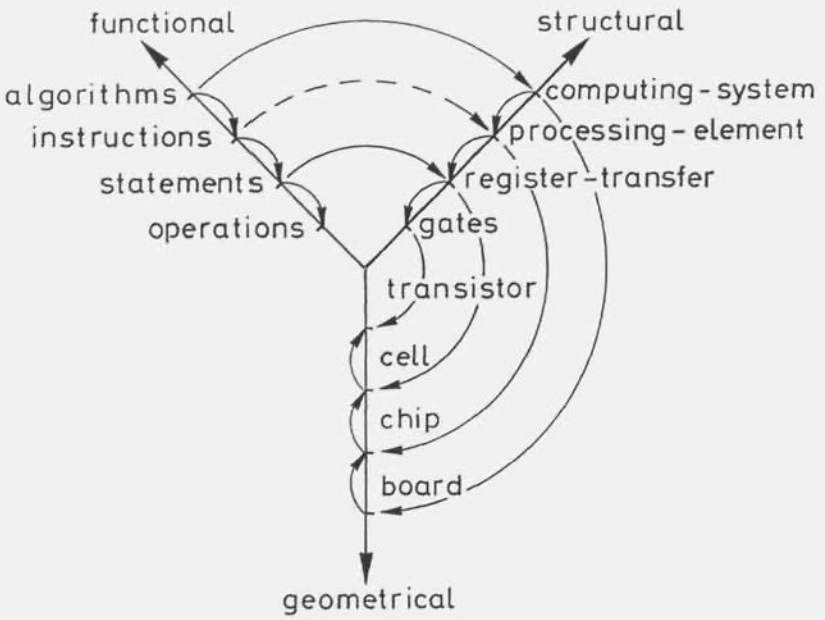

Figure 3.2. Y-chart

the function to its arguments. In order to verify the implementation of a function, we need a method for defining its abstract meaning.

A different type of abstraction is type- or data abstraction. The term "abstract data-type" is used to refer to a class of objects defined by a representation independent specification. The large number of interrelated attributes associated with a data-object may be separated according to the nature of the information that the attributes convey about the data objects that they qualify. Two kinds of attributes are:

1. those that define the representation of objects and the implementation of the operations associated with them, in terms of other objects and operations.

2. those that specify the names and define the abstract meaning of the operations associated with an object.

In the course of a design one is concerned mostly with the attributes of point 2. The user of a data object should not be interested in its representation, nor should he need to know details of the implementation of the operations in order to invoke these. The class constructs appearing in many so-called object-oriented languages [Gold83, Cox86], offer a mechanism for binding together the operations and storage structure representing a type. The class construct used in these languages does not however offer a representation independent means for specifying the effect of the operations.

In order to compare different models and languages, the following criteria are introduced:

Simplicity

The model and/or language should be easy to learn and use. It should be conceptually close to the intuitive model used by "expert" designers. In addition, a simple language increases the possibility of defining a formal semantics.

Expressive power

The expressive power of a language is dependent on its abstraction mechanisms and the build-in constructs. Build-in constructs add to the complexity of the model/language; there will usually be a trade-off between expressive power and simplicity.

Mathematical tractability

In practice this implies that the formal semantics must be sufficiently simple to allow effective algebraic manipulations. The question of mathematical tractability is also of utmost importance for verification. Only when different specifications and/or implementations can be mapped to one underlying language, e.g. the language of first order logic, or when we can reason about programs using the laws of an algebra of programs as shown by Backus [Back78], is it possible to verify the equivalence of the

behaviors of the different specifications / implementations.

Regularity

The language or model must have adequate facilities for describing regular structures, since the algorithms discussed in chapter two require a regular architecture.

Locality of interconnections

In order to be able to design an optimal architecture, or to define an optimal mapping of the algorithm on a VLSI chip, the number of non-local interconnections must be minimal. Therefore, the model or language must be detailed enough to be able to determine whether an interconnection is local.

3.2 Models and Languages

In this section we introduce a number of models and languages that were developed in order to simplify the design, implementation and analysis of software and hardware systems. Due to the nature of hardware systems and the increased complexity of software systems, we can model both as a collection of interacting modules or subsystems. We will therefore not distinguish between hardware and software systems, since it is only at the implementation level, respectively the mapping to hardware modules and the translation to instruction sequences interpretable by a particular processor, that the differences become relevant. This does not affect the fact that the limitations of a particular technology will influence design trade-offs at higher levels of design, as discussed in section 3.1. In our view such influences will always remain and are in fact essential in order to be able to design efficient systems.

The model underlying conventional programming languages, such as C, Pascal and Fortran, is based on the so-called Von Neumann model, i.e. a single CPU reads instructions and data from a memory over a one word wide bus. This inherently sequential model is not very suited for the design and specification of concurrent systems, as mentioned above. In

addition, conventional languages usually have a complicated semantics, because one can modify the state of a computation one word at a time, e.g. by assigning a new value to a memory word. The semantics are complicated, because in general it is very difficult to track all places where a variable, the abstraction of a memory word, may change value, due to the presence of so-called side-effects. Side-effects can occur when two or more variables refer to the same memory area. A typical example is when a procedure changes the value of a variable that was not declared within its body, e.g. when one or more of its arguments are passed by their addresses (call-by-name). For this and other reasons, conventional languages have complicated semantics and are difficult to prove correct [Back78].

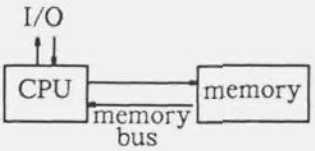

Figure 3.3. The Von-Neuman model of computing

The above two problems have lead to two different, but interrelated developments:

- Development of *process oriented models* for modeling parallelism.
- Development of *applicative languages* with a simple semantics that can be manipulated algebraically.

3.2.1 Process Oriented Models

The development of process oriented models finds its root in the development of complex operating systems for time shared computer systems in the 1960's. The specification and design of an operating system consisting of many interacting activities naturally leads to the adoption of a process oriented model of computation. The process oriented approach makes it possible to decompose a task into a number of subtasks each of which can be independently specified. The processes communicate via channels and are synchronized using special synchronization mechanisms, e.g. semaphores. The UNIX operating system for example is composed of a

large number of processes for controlling different resources, e.g. printers and terminals.

A formalization and extension of this work can be found in the model of Communicating Sequential Processes (CSP) [Hoar85].

Another tool for modeling systems with interacting concurrent components are Petri nets. Petri nets are an important tool for the study of various properties of a system. There are several ways in which Petri nets can be used in the design and analysis of a system. First it is possible to model one or more aspects of a system, that has been designed in another methodology, with a Petri net, which can then be analyzed. Any problems encountered in the analysis can then be traced back to the design, remedied and the modified design again be modeled and so on. Another approach is that the entire design and specification/implementation process is carried out in terms of Petri nets. Petri nets are discussed further in section 3.2.1.2.

3.2.1.1 CSP

The computational model proposed by Hoare [Hoar85], develops the view of a computational system as a network of *Communicating Sequential Processes* (CSP), each of which is characterized by its externally observable behavior, i.e. by the actions or events in which it is prepared to engage. Since there is no fundamental distinction between a process and its environment, the boundary between the two can be drawn arbitrarily; the model provides a unified method for modeling computational systems, including interaction with their environment.

A process is defined by its behavior. The behavior of a process is defined by the set of events in which it is prepared to engage at any point in its evolution †. In Hoare's terminology a particular evolution is described by a trace. The set of all possible traces of a process defines the behavior of the

† We will refer to the succession of events in which the process engages as the evolution of the process.

process ‡.

Processes are defined using some simple notation:
A process that first engages in the event x and then behaves exactly as another process P is described with the prefix notation as follows:

$$(x \rightarrow P) \tag{3.1}$$

Repetitive behavior patters are described with the use of recursion. For example, a simple clock that does nothing but tick, is described by the equation:

$$CLOCK = tick \rightarrow CLOCK \tag{3.2}$$

or

$$CLOCK = \mu X.(tick \rightarrow X) \tag{3.3}$$

which says that CLOCK is the solution of the recursion (3.2) or (3.3). Since (3.2) and also (3.3), have the property of being guarded, the solution is guaranteed to be unique, due to the fixed-point theorem [Hoar85, p. 96].

By means of prefixing and recursion, we can describe processes that exhibit a single possible stream of behavior. In order to describe processes that will allow their behavior to be influenced by their environment, Hoare introduces the choice operator. If x and y are distinct events then:

$$(x \rightarrow P \mid y \rightarrow Q) \tag{3.4}$$

describes a process that initially engages in either of the events x and y and subsequently behaves as either P or Q, depending on which choice occurred.

Based on the notation introduced until now, it is already possible to introduce a number of laws that allow us to reason about the behavioral

‡ Actually Hoare shows that there is a one-one correspondence between each process P and the pairs of sets (αP,traces(P)) where αP is the set of events in which the process is actually capable of engaging, the so-called alphabet of the process.

equivalence of processes. For example, two processes defined by choice are different if they offer different choices on the first event, or if after the first event they behave differently.

A very interesting possibility is also that we can, in general, verify whether a process P "satisfies" a specification S. In CSP specifications take the form of predicates that state properties that all traces of a process P have to adhere to. Hoare derives a collection of laws that permit the use of mathematical reasoning to verify that a process P meets a specification S.

When a process offers a choice of events, the choice which event will actually occur is controlled by the environment within which the process evolves. Since the environment can be defined as a process itself, this leads us to consider the behavior of a system composed of, potentially many, processes evolving concurrently. The interactions between these processes may be regarded as events that require simultaneous participation of all processes involved. The notation (P ‖ Q) denotes the process which behaves like the system composed of P and Q.

The next step is the introduction of non-deterministic processes. A non-deterministic process, as defined by Hoare, is a process in which the environment can't observe or control the choice between events, although the particular choice may be inferred from the subsequent behavior of the process. Non-determinism is useful in maintaining a high level of abstraction in the description of physical systems. The main advantage is *that a process description may be deliberately vague. The process specified* by (P □ Q), where □ is the non-deterministic choice operator can be implemented either as P or as Q. The final choice may depend on criteria that are irrelevant for the specification.

Input and output are defined by extending the notation to make it possible to associate variables and expressions with events. Variables are associated with input events; expressions are associated with output events. A simple incrementer, i.e. a process that inputs a value and then outputs the same value incremented by one, is defined in CSP as follows:

$$INCR=\mu X.(in?var\rightarrow out!(var + 1)\rightarrow X) \qquad (3.5)$$

The last major step is the introduction of sequential composition of processes, which allows the definition of control structures similar to those in conventional languages, e.g. if-then-else, while-do etc.

3.2.1.2 Petri Nets:

Petri nets and Petri net theory [Pete81], form a valuable tool for modeling and analyzing systems composed of potentially many interacting and simultaneously active components. In this section we will give an overview of Petri nets.

Structure

The structure of a Petri net is defined by its places, its transitions, input function and output function. The input and output functions relate transitions and places.

Definition [Pete81]

A Petri net structure C is a four tuple $C = (P,T,I,O)$. $P = \{p_1,p_2,...,p_n\}$ is a finite set of places. $T = \{t_1,t_2,...,t_m\}$ is a finite set of transitions. The set of places and the set of transitions are disjoint. $I : T \rightarrow P^{\infty}$ is the input function, a mapping from transitions to bags of places. $O : T \rightarrow P^{\infty}$ is the output function, a mapping from transitions to bags of places.

A Petri net structure can be represented by a *bipartite, directed multigraph* having two types of nodes corresponding to the places and transitions of the Petri net structure. Directed arcs connect the places and transitions. A marking μ is an assignment of tokens to the places of a Petri net. A token is a primitive concept of Petri nets. Tokens reside in places and control the execution of the transitions of the net. A Petri net executes by *firing* transitions. A transition fires by removing tokens from its input places and creating new tokens which are distributed to its output places. As a result the number and position of tokens in a Petri net may change during the execution of the net. The state of a Petri net is defined by its marking. The firing of a transition represents a change of state of the Petri net by a change of its marking.

Given a Petri net C = (P,T,I,O) and an initial marking μ^0, we can execute the Petri net by successive transition firings. Two sequences result from the firing of the Petri net: the 'sequence of markings' $(\mu^0,\mu^1,...)$ and the 'sequence of transitions' that were fired $(t_{j_0},t_{j_1},...)$. Based on this the reachability set $R(C,\mu)$ of a Petri net C with marking μ can be defined as the set of markings reachable from μ. A marking μ^i is reachable if there exists a set of transition firings which will change μ into μ^i.

Modeling

The usefulness of Petri nets for modeling systems derives from the fact that many systems can be modeled as performing a sequence of actions whose occurrence is controlled by a set of conditions. The set of all conditions can be viewed as defining the state of the system. This view of a system directly corresponds to a Petri net. Conditions are places; a condition is true if the place contains one or more tokens. Transitions are events; the inputs are the preconditions, the outputs are the postconditions of the event. The usefulness of Petri nets is proven by the large number of applications that can be modeled by them, including computer hardware and software. Petri nets can be used to precisely model e.g. parallelism and the synchronization problems it poses, e.g. in the case of shared resources.

Analysis

In order to gain insight in the behavior of a Petri net, it is necessary to analyze it. Important properties that can be determined are: *safeness*, *boundedness* and *liveness*.

Safeness is a special case of boundedness. A place in a Petri net is k-safe or k-bounded if the number of tokens in that place can not exceed an integer k. A Petri net is said to be k-safe if every place is k-safe. A place that is 1-safe is simply called safe.

Conservation is a property that is used to prove that tokens that represent e.g. resources are neither created nor destroyed.

Liveness is an important property that can be determined to make sure that the Petri does not contain deadlocks. A transition in a Petri net is live if it

is possible to find a sequence of transition firings that take the Petri net from its current marking to one in which the transition is enabled. A transition is deadlocked if it is not life.

Most of the analysis problems are concerned with reachable markings. Consequently the major analysis techniques for Petri nets are based on construction and analysis of the so-called reachability tree, which is a finite representation of the set of reachable markings of a Petri net.

3.2.2 Applicative Languages

The development of applicative languages was motivated in large part by the desire to provide a more rigorous mathematical basis for programming. The lambda-calculus provides such a basis and lies at the root of the development of LISP. Although many LISP dialects offer a variety of non-applicative constructs, such as assignment, the power of the language is derived in large part from its applicative kernel.

3.2.2.1 FP and AST systems

In his often referenced paper "*Can Programming be Liberated from the Von Neumann Style ? A Functional Style and its Algebra of Programs*" Backus [Back78] reviews the deficiencies of existing programming languages, motivated by the Von Neumann model of computing and proposes a functional style of programming that allows mathematical reasoning methods to be applied to programs. In his FP language, Backus reduces 'programming' to algebraic manipulations of programs that represent functions. A program is an expression that consists of functional operators, the so-called combining forms and (names of) functions. The combining forms create new functions using other, previously defined and named, functions. The algebra of programs allows the formulation of laws that are useful in reasoning about and/or proving properties of programs.

In an FP system one can *apply* any function to a sequence of input values†.

If the structure of the input sequence matches with one that can be handled by the function, the function will compute the desired result; otherwise it will return a special error value to indicate failure.

FP systems have a set of predefined functions, that can be classified as follows:

- sequence manipulation, e.g. *head* and *tail*
- arithmetic, e.g. $+$, $-$, $\times$ etc.
- predicates, e.g. the relational operators, $\leqslant$, $\geqslant$, $\equiv$ etc.

Each of these functions expects its arguments to be mapped on the elements of an input sequence in a particular fashion.

Examples of functional forms are:

- $\circ$, composition. The composition of two functions f_1 and f_2, denoted by $(f_2 \circ f_1)$ is a function that applies f_2 to the result of applying f_1 to the input sequence
- [,], construction. The construction of two functions f_1 and f_2, denoted by $[f_1,f_2]$ is a function that returns a sequence that consists of two subsequences; the first being the result of applying f_1 to the input sequence, the second the result of applying f_2 to the input sequence
- α, apply. The application of a function f, denoted by αf, is a function that applies the function f to each element of the input sequence it is applied to.

Using the functions and functional forms as introduced above, a FP programmer can define new functions. For example, a function that does multiplication by adding the logarithms of the elements of its input

† Note that an input sequence may contain subsequences.

sequence, is defined as follows:

$$\text{def MULT} = \exp \circ /_L + \circ \log \tag{3.6}$$

The definition can be read as follows: Apply 'log' to every element of the input sequence, sum the resulting values, i.e. distribute plus, using the functional form $/_L$ and take the exponent of the result.

A disadvantage of applicative languages is that they can't be used for the description of history-sensitive systems. Since practical systems are almost always history sensitive, this restricts the usefulness of purely applicative languages. In order to describe history sensitive systems Backus introduces so-called Applicative State Transition (AST) systems. An AST system combines an applicative style of programming with a state-transition semantics. The problems associated with defining a clear and simple mathematical semantics for conventional, imperative programming languages show however that it is necessary to restrict the number of state transitions. Programs written in conventional languages are not suitable to mathematical analysis because of the large number of assignment statements. An assignment statement changes the state of a computation by changing the value associated with a variable in a particular environment. In a conventional programming language the state can change while evaluating a function, or block of statements. Backus, in defining AST systems, didn't allow this. The state of an AST system is changed only once per major computation cycle, in the sense that for every input, the AST system computes an output and a new state (cf. Figure 3.4). The new state replaces the old state on the subsequent input. New states and outputs are computed by a functional program.

According to Backus, a reasonable AST system should have the following properties:

- State transitions occur only once per major computation cycle.
- Programs are written in a functional language.
- There are no side-effects, i.e. during the evaluation of a function the state may not change.

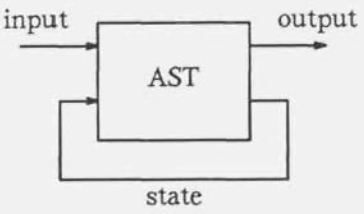

Figure 3.4. AST system

- The framework consists only of: (a) the syntax and semantics of the applicative subsystem and (b) the semantics of the state-transition mechanism.

Backus distinguishes two types of applicative systems. FP systems are characterized by a fixed set of functional forms. The language does not allow the definition of new functional forms. FFP systems on the other hand, do allow this. This is accomplished by denoting functions by objects, using the representation function ρ and by introducing expressions that are to be evaluated using the meaning function μ. FFP systems are more powerful than FP systems.

3.2.2.2 μFP

The language μFP, developed by Sheeran at Oxford University [Shee83], is an extension of Backus FP language. In μFP, a function f takes a sequence of inputs and produces a sequence of outputs. The semantics of μFP is defined in terms of FP with the help of a meaning function M. For example, the meaning of a function 'f', which contains no internal state†, is just αf (in FP), where α is the FP functional form introduced above.

$$M[[\ f\ ]] = \alpha f$$

As a result, in μFP every function f works in a repetitive manner on a sequence of inputs, giving a sequence of outputs.

One of the major extensions of μFP over FP is the definition of functions with an internal state. By applying the functional form μ to a function f mapping pairs of inputs to pairs of outputs, one of the inputs is connected to one of the outputs, creating an internal state.

Consider for example the (μFP) function '[2, 1]', i.e. the construction of the selectors '1' and '2'. This function exchanges the first and second element of each pair of inputs, as shown in Figure 3.5.

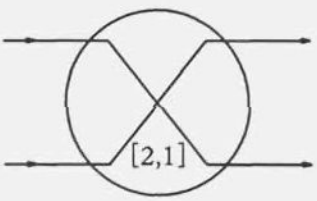

Figure 3.5. '[2,1]': a μFP function exchanging the elements of its input

The function '[2, 1]' can be transformed into a simple shift register cell by applying μ to it. The second element of the output is fed back to the second input through a memory element that provides a delay (see figure 3.6). The sequence on the second input/output is transformed into the state.

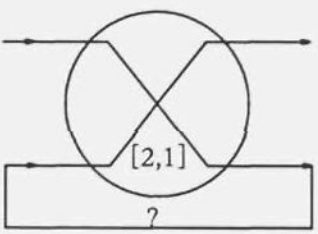

Figure 3.6. 'μ[2,1]': a simple shift register cell

† The internal state of a function is introduced by the functional form μ.

The initial value of this state sequence is assumed to be '?', the don't care value.

One of the assumptions underlying the μFP language is that for most systems a high level specification of the form 'μ[f, g]' can be found. Such a description need not be suitable for a direct implementation however, because of the complexity of the f and g functions and/or the complexity of the state. In order to find a suitable implementation the system 'μ[f, g]' has to be decomposed into small easily implementable functions, e.g. of the type shown in Figure 3.6. The process of transforming from specification to implementation can be viewed as one of "pushing" μ's further and further down into the μFP expression, which becomes more and more complicated [Shee83, p. 20]. Part of this complexity can be removed by defining suitable abbreviations for parts of the expression, e.g. by giving meaningful names to complicated compositions of selector functions.

μFP allows a large number of algebraic laws to be formulated that are useful in proving properties of (μFP) programs. The correctness of these laws themselves is mostly proved in the underlying FP system, using the meaning function M.

3.2.2.3 SILAGE

SILAGE [Hilf85], is a simple applicative language for high level description of signal processing algorithms. It attempts to capture the flavor of graphical signal-flow representations often used to represent these algorithms, with an applicative notation.

A design description consists of a set of equations, relating the values of the inputs, outputs and intermediate values of functions as if they were static, timeless quantities. In fact however, all quantities in SILAGE are infinite arrays indexed by an integer quantity that one can think of as "time" or "sample number". Previous values are accessible via the operator @. The notation 'x @ n' denotes the value of x n samples ago. An equation or definition of the form: 'name = expression', defines a symbol or array element name to have the value indicated by the expression. The syntax for expressions is much the same as for a conventional language. The primary

abstraction mechanism, as could be expected from a functional language, is functional abstraction. A function itself is defined by a set of equations. Functions may return multiple results and can't be recursive.

SILAGE is intended as the input language for a high-level silicon compiler, which explains its simple control structure and the lack of assignment statements. The CATHEDRAL-II system, under development at Leuven University [Raba85] also makes use of SILAGE. SILAGE allows hints for silicon compilers and other synthesis tools, to be included in the system description in the form of so-called "pragma" statements. These pragma statements may for example assign an expression to a particular hardware unit (processor), thereby simplifying the task of mapping the algorithm on a hardware architecture.

3.2.2.4 System Semantics

Boute [Bout86], describes an approach to system design that is based on an extension of the denotational theory of programming languages to the description of arbitrary systems. The denotational semantics of a programming language associates a meaning, in general in the form of a mathematical object, with every construct in the language. The meaning of a sentence is derived from the meaning of the constructs forming the sentence and the method by which they are combined. The mathematical object denoting the meaning of a sentence or construct can be manipulated according to the laws of the domain in which it is defined, in order to prove various properties about a program.

System semantics defines meaning functions for the various properties of interest in a physical system. Every meaning function relates the description of the system to the value of a property. Boute describes a generic function language (SFGL) that can be filled in to suit various purposes. The language SBFL for example is used for describing combinatorial digital circuits. The approach described by Boute is to define the syntax such that the compositionality of the semantic definition, i.e. the ease by which the meaning of a composite construct is expressible in the meanings of its constituents is optimized.

The interesting concept of system semantics is that a single description is sufficient to derive all properties of interest. Different properties can be found by evaluating their corresponding meaning functions in the domain of interest.

3.2.3 Design Languages

In this section we will discuss a number of languages that combine elements of applicative and process oriented modeling.

3.2.3.1 DSM

Cremers and Hibbard [Crem85], present a programming notation for locally synchronized algorithms to be implemented on a locally interconnected static structure of asynchronous processing elements. This particular architecture was chosen so as to match the technological constraints imposed by VLSI technology on interconnections and system-wide clocking. The notation is based on the principles of AST systems [Back78] and data-flow computation. The formal notion underlying the definition of the AST system is that of a data-space [Crem76]. A data-space consists of two parts: a transition system and an information structure. The transition system defines the processor p, as a relation on a set of states X. An information structure for X is a set of functions {F}, such that each member $f \in F$ is totally defined on X. A triple $D = (X,F,p)$ is termed a data-space if it satisfies a number of axioms that ensure the consistency of the data-space.

Cremers and Hibbard describe the application of data-spaces to the executable specification of algorithms. This requires syntax for specifying both structured state spaces (X,F) and a transition system (X,p). The state space can be defined as a collection of cell declarations. A cell is a pair of the form (nameType, contentsType) and represents a data defined function on X. To evaluate it, one simply looks up the value associated with the name of the function in that state.

The processor is defined by a functional program consisting of a single expression whose only argument is the state. The result of evaluating the expression against the state is a set of name-contents pairs. During the

computation of that set there can be no side-effects, as required by the Applicative State Transition principle. The set of pairs computed is applied at once to the state, thus giving a new state. Subsequently, the processor function can be evaluated again. The expression defining the processor function can invoke auxiliary functions. These auxiliary functions are otherwise similar to the processor function and make it possible to specify the processor function in an hierarchical fashion.

To construct a network of data-spaces Cremers and Hibbard introduce a simple syntactic construct to encapsulate a data-space definition and give it a name. Any cells defined within the data-space, with the exception of so-called synchronized cells, are strictly local to the subspace. Synchronized cells allow information exchange between otherwise independent subspaces. To implement this, a synchronized cell holds, in addition to the data it is declared to hold, a status which is either "ready to read", or "ready to write". Equivalency of synchronized cells explicitly establishes the interconnection structure between the subspaces of a data-space. The interconnection structure is static; it can't change during the execution of a data-space.

3.2.3.2 SIGNAL

SIGNAL [Guer85] is a data-flow oriented language for signal and image processing developed at IRISA/INRIA. The language aims at the description of real-time synchronous systems. Real-time refers to the capability of the system to respond to externally generated input stimuli with a finite and specifiable delay. Synchronous refers to the fact that all activities occurring in a system defined by a SIGNAL description can be totally ordered.

SIGNAL describes a system as a static network of hierarchically specified processes. The processes are ultimately composed of primitive processes, so-called generators. The network is constructed using a set of structural operators, that allow processes to be constructed from subprocesses in an hierarchical manner. The structural operators connect, rename and/or hide inputs and outputs of processes. Interconnections between processes are made based on the equality of names of inputs and outputs. SIGNAL provides a complete set of structural operators so that any static network of processes can be constructed.

Generator processes are distinguished in functional and temporal generators. Functional generators are defined by expressions. Temporal generators are a unique feature of SIGNAL and describe the timing relationships between the different signals in a process. The temporal generators form the basis of a clock calculus that is used to verify the correctness of the timing relationships between all signals in a process.

3.2.3.3 CRYSTAL

The language CRYSTAL defined by Chen [Chen83] uses a different approach to define the semantics of an algorithm. An algorithm describing a system is defined as a set of recursion equations in the space-time domain. The least-fixed point of these equations is then taken as the "meaning" of the algorithm.

The model underlying this language is that of a collection of processes, where each process consists of:

- control state register
- data store
- machinery for computing a state-transition function
- input/output ports.

Each process is located in a space coordinate system. The ordering of operations in a processor introduces a (local or global) time coordinate. Relationships between processes are established by identifying input and output ports of processes in the space time coordinate system.

State-transition functions are the basic units for constructing systems. Chen shows that by choosing the primitive state-transition functions appropriately, it is possible to cover a wide range of system description levels, extending from switch-level models to systolic and wavefront array type models.

3.3 Discussion

The decomposition and abstraction mechanism available in a model or a language, determine to a large extent its usefulness as a design language; the composition mechanisms determine the kind of 'structures' that can be formed, the abstraction mechanisms determine the complexity of the behavioral descriptions associated with the modules.

The composition mechanisms in the language CSP are used to define the behavior of a process in terms of the behavior of the subprocesses. The behavior of a subprocess is simpler because it hides the communication between the subprocesses. The behavior of a process is explicitly described by its traces. Properties of a process can be established by analyzing its traces. The trace set of a process does not specify how the process is implemented in terms of subprocesses. In this sense, the trace can be viewed as an abstract specification of the process. The abstraction of a process as an independent computational activity makes CSP very suited for description of complex software systems, e.g. communication protocols and operating systems. CSP allows manipulation of both traces and programs.

Decomposition and abstraction mechanisms are much harder to recognize in Petri net models. Except from the fact that a Petri net model offers a high degree of formalization in the modeling of a system, Petri nets do not include concepts for defining Petri nets in terms of themselves, e.g. as a composition of Petri nets. Petri nets, without further restrictions, can also not be viewed as e.g. transitions and included as such in another Petri net. It is however possible to restrict Petri nets in such a way that they can be composed. Typically this is done by defining a language whose semantics is defined in terms of Petri nets [Ramm87]. The primitive constructs in such a language have equivalent Petri net models. By properly restricting the primitives and the way in which they can be composed, it is moreover possible to guarantee certain properties of the Petri nets thus created, e.g. liveness and boundedness. An interesting aspect is that because of the broad application domain of Petri nets, one can define a design language that covers a wide spectrum of abstraction levels (cf. section 3.1).

Applicative languages are defined in large part by the so-called combining forms, i.e. the composition constructs. The abstraction mechanism underlying applicative languages is that of a function, mapping an input value or a tuple of input values to an output value or tuple of such values. The usefulness of applicative languages for design derives from two properties:

1. it is possible to associate a 'structure' with a composition of functions
2. functional programs can be transformed according to a set of laws of an underlying algebra of programs.

The interesting aspect is that the 'structure' or another property of a functional program can thus be optimized without changing the function [Bout86].

The design languages discussed in section 3.2.3 combine aspects of the two main approaches, i.e. process oriented modeling and function composition as used in applicative languages.

The DSM model and language of Cremers and Hibbard combines an applicative framework with a state transition semantics. A system is modeled by a data-space and a processor function. The processor function is applied to the state defined by the data-space and consists of a set of statements that define its behavior. Statements may call other functions, that are defined in a similar fashion. A distributed implementation may be defined by identifying 'local' states, i.e. collections of statements acting on a subset of the state. The local states may be encapsulated in local data-space, with their own processor function. The communication between data-spaces is by means of 'synchronized' cells, that implement a form of asynchronous single token pass mechanism. The extended model, i.e. the one including the concept of local data-spaces, can be used to define regular processor architectures, if these spaces can be parameterized and instantiated with some kind of iterative construct, as for example suggested in Cremers and Hibbard [Crem85].

The language SIGNAL takes a different approach, in that a system is defined by a sequence of successive decompositions of modules. Modules are

specified by defining their structure. The abstract behavior of a module can not be defined in SIGNAL. In order to verify the consistency of a module it is therefore necessary to know its final decomposition in terms of so-called primitive generators.

Verification

The kind of verification that can be done on a design depends very much on the model underlying the language in which the design is expressed. In general we may distinguish between different types of verification:

1. verifying whether an implementation description correctly implements a specification
2. verifying whether two implementation descriptions implement the same specification, i.e. whether they are equal
3. verifying whether different properties of the system defined by an implementation fall withing a certain range.

In order for a model to allow verification ad. 1, it is necessary that it allows the definition of the abstract behavior of a module, e.g. in the form of a 'specification'. From the models discussed in section 3.2, only the CSP language allows this, in the sense that we can reason over the traces of a process and that we can verify that a particular process does generate a particular trace-set. A unique feature of SIGNAL is the fact that it allows complete verification of the timing relationships between the various signals occurring in a system. This is achieved by defining a clock-calculus, based on the timing relationships that exist between the signals entering and leaving the primitive generators.

Both CSP and applicative languages allow verification ad. 2. To make this verification possible, it is necessary that the model allows the definition of a body of laws to transform design descriptions, while leaving the behavior or some other property invariant. The verification of the equality of two design descriptions then amounts to finding a sequence of transformations that transform one description into the other.

Verification ad. 3 requires the existence of a well defined relationship between the design description and the property of interest. All languages and models allow this type of verification; they differ only in the properties that can be determined. In fact, the approach taken by Boute [Bout86], can be seen as a formalization of this type of verification, in the sense that the semantic domain of the design description is chosen as the domain in which the property of interest is defined. Other models allow only certain types of properties to be determined. For example, Petri nets allow properties like liveness and boundedness to be determined, while CSP programs allow specification of properties of traces, e.g. the number of occurrences of a particular event. A special kind of verification is allowed by the language CRYSTAL [Chen83], which allows the least-fixed point of the system, defined by a CRYSTAL program, to be computed.

4. HIFI: Design Method and Computational Model

In this chapter we will give an overview of the HIFI design method proposed by us. HIFI is an acronym for Hierarchical Interactive Flowgraph Integration, and is tailored towards the design of VLSI array processors, specifically the systolic and wavefront arrays discussed in chapter 2. The name HIFI was chosen to emphasize the hierarchical and interactive nature of the design process and to stress the importance of a graphical user interface based on signal/data flowgraphs.

The HIFI system aims at the initial phases of a design, mainly the translation from a numerical problem to a concurrent algorithm, and the subsequent mapping of this algorithm on either dedicated VLSI hardware or a programmable multiprocessor system, such as the Intel iPSC hypercube [Inte86], or a network of transputers [Inmo86]. The HIFI system does not provide support for the actual design, e.g. routing and placement, detailed simulation and so on, of the VLSI processor elements required to build such a multiprocessor system, although it can be integrated into a more general VLSI design system [Dewi86].

The main emphasis in the HIFI system is on the systematic derivation of regular multiprocessor systems for implementing certain classes of signal and image processing problems (see chapter 2), where the constraints imposed by a VLSI implementation, i.e. regularity and locality of interconnections, are observed. Furthermore, *HIFI* is intended as a *design system*, not just a tool for describing a design at various levels of detail, but a tool that : (1) assists the designer in the transformations required to convert an initial given description into a series of more detailed descriptions, (2) allows mapping of these successively more detailed, i.e. less abstract, descriptions onto dedicated or programmable multiprocessor systems, (3) provides persistent storage, i.e. a database and (4) provides a user interface to retrieve, create and modify the objects that are in the database.

In this chapter we will give an overview and discuss the computational model underlying the HIFI design methodology.

4.1 A quick overview

The principal abstraction upon which the HIFI method is build, is that of a function; a mapping from inputs to outputs. When considered at a sufficiently high level of abstraction, any signal processing system can be described by its input-output map, i.e. as a function F from inputs, viewed over all space and time, to outputs, similarly viewed over all space and time. Except for the simplest systems however, it is not feasible to implement the input-output or I/O function of a system directly.

In the sequel we will give a systematic discussion of methods to decompose F, in order to obtain a system that can be implemented. Broadly speaking these (decomposition) methods can be brought into four categories based on (1) the domain in which the decomposition takes place, i.e. either space or time and (2) regularity, i.e. a decomposition can be either regular or irregular. Decompositions are specified by means of *Signal Flow Graphs* (SFG). A SFG is a directed graph that consist of nodes and edges. Each node has a set of inputs and outputs. The edges are used to specify the interconnections between these inputs and outputs. Edges do not have a functionality of their own. A decomposition in the space domain is represented by the interconnection structure of a SFG. Decomposition in the time domain is represented by the state-transition mechanism associated with the nodes of a SFG. A regular decomposition in space is specified by a *regular* SFG, i.e. a SFG whose nodes can be mapped on the grid-points of an n-dimensional space. In fact the nodes in a SFG may perform any one from a collection of functions, each function determining its successor function, i.e. the function to be performed next by the node, according to a mechanism similar to that of the AST systems discussed in section 3.3.1.

Consider for example the SFG shown in Figure 4.1(a). It represents a Finite State Machine (FSM) that is implemented by two functions, f and g. The SFG contains two nodes; the first one implements the functions f and g, the second one is a so-called delay node, which is used to represent the state.

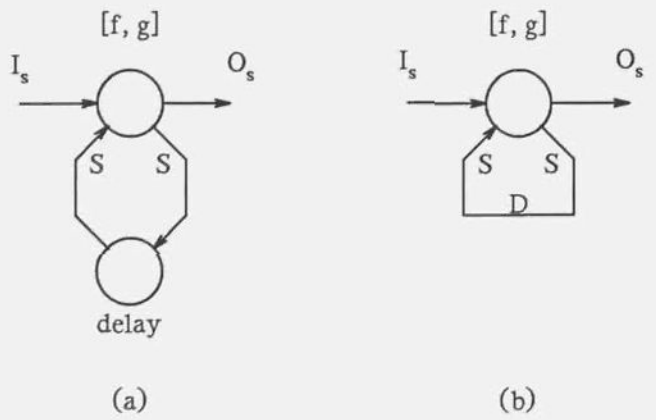

Figure 4.1. Finite State Machine represented by a SFG

The function g maps input and current state to the next state, i.e. $g:(S \times I_s) \rightarrow S$. The function f maps either the current state to the current output, i.e. $f:S \rightarrow O_s$, in which case the FSM is a so-called Moore machine, or it maps the current state and the current input to the output, i.e. $f:(S \times I_s) \rightarrow O_s$, in which case the FSM is a so-called Mealy machine. The first node simultaneously applies the functions f and g to the input data. The delay node is more complicated. It represents a collection of data-defined functions $\{f_\alpha\}$ that correspond to the collection of possible states of the FSM. Initially the delay node performs a function f_0, corresponding to the initial state, say $state_0$, of the FSM. This function outputs the value '$state_0$' on the output edge of the delay. Next it performs a function that reads the value of the state, as computed by the function g, from the input edge of the delay. Subsequently the delay node performs a function f_α that outputs the value of the state read from the input edge of the delay. The next function performed by the delay is again a read function etc. A detailed definition of the operation of nodes and SFG's will be given in the next section (section 4.2). To reduce the work involved in specifying a SFG, we will introduce special icons for representing certain often occurring nodes, such as delays. In the case of a delay, we will simply put a letter 'D' along the edge, as shown in Figure 4.1(b). A complete specification of the

FSM requires the functions f and g to be defined. The HIFI system will allow definition of f and g either by decomposition or by an executable specification, e.g. in the form of a procedure that can be evaluated to compute the output value of the function, given the input values of the function,.

In order to simplify the definition and interpretation of SFG's specified graphically, we will define a restricted type of SFG, i.e. a *Dependence Graph* (DG). The restrictions will concern both the computational model underlying the nodes and the structure of the SFG (cf. section 4.2.). In essence, the nodes in a DG represent stateless functions. As a result, the operation of a DG can be understood more directly from its representation as a SFG, where the function performed by a node may depend on its history.

An overview of the main concepts in the HIFI system is shown in Figure 4.2. The basic sequence of steps is:

- **define** a function, i.e. a relation between inputs and outputs (cf. Figure 4.2(a))
- **refine** the function, i.e. define a Dependence Graph (cf. Figure 4.2(b))
- **partition** the DG, i.e. map it on a processor architecture (cf. Figure 4.2(c)).

The definition of SFG's will be simplified if we have a computational model that can be applied uniformly. Such a model will be discussed in section 4.2. Characteristic for the model discussed there is the adoption of a uniform communication mechanism. The modeling power of a SFG then becomes dependent on the model associated with a node. To increase the flexibility of the design system, we will define not a single model for a node, but a hierarchy of successively more specialized models, such that the more general models include the more specialized ones. The models that we will define range in complexity from CSP processes [Hoar85], the most general, via AST mechanisms, to a purely functional model.

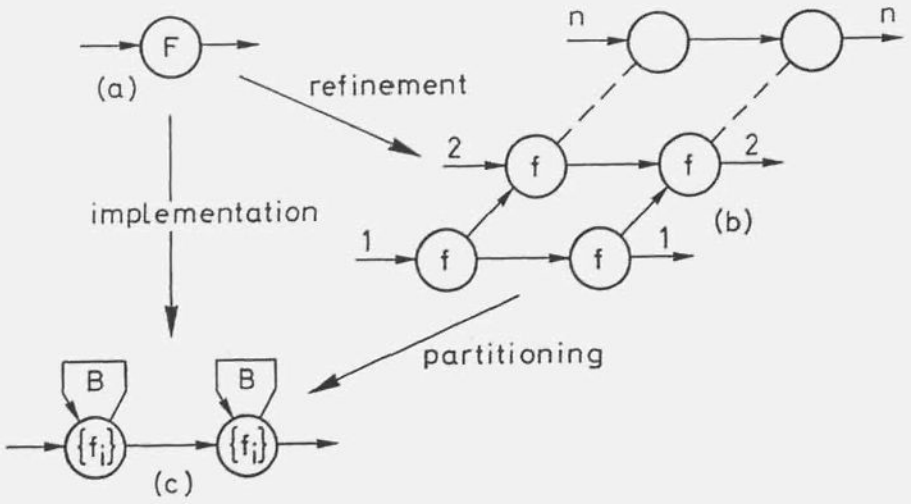

Figure 4.2. HIFI: design steps

An important aspect of the HIFI methodology are its recursive decomposition methods. These allow a function to be implemented by a composition of (sub) functions, each of which can be implemented by a composition of (sub-sub) functions etc. This property allows an easy combination of top-down and bottom-up design hierarchical design styles. Function decomposition will be discussed in more detail in chapter five.

The second major part of the HIFI system provides the tools for defining an implementation of a function, based on its decompositions. The designer can define "nodes" that can perform any one from a number of functions, according to the values on their control inputs. The mapping from a decomposition to an implementation is considered independent of the decomposition, to allow a designer to define different implementations taking into account criteria such as the available hardware, throughput demands, timing discipline etc. The mapping process above is referred to as the "partitioning phase" (cf. Figure 4.2(c)). The result of the mapping is represented by a *Signal Flow Graph*. The SFG in Figure 4.2(c) is also referred to as an "implementation" of the function F. Partitioning is discussed in more detail in chapter five.

Another aspect is the definition of the data-types associated with the edges connecting the nodes. Not only will we have to define the types themselves, it will also be necessary to introduce some kind of *algebra of types*, in order to relate and/or verify the types associated with the input and output edges at various levels of refinement of a function. Types will in fact provide an important mechanism for parameterizing refinements, as suggested by Figure 4.2 (a) and (b). The single input of the function F gets decomposed into an array of inputs. It is assumed that the type of the input of F is such that it can be decomposed in a similar fashion. For example, if the type were say 'vector of integer' the inputs in Figure 4.2(b) would all be of type integer. The number of inputs would be determined by the number of elements of the vector. Subsequent refinements could carry this further. For example, if the function f adds two integers, then it could be decomposed in an array of full adders. The number of full adders and the number of inputs of the refinement would be determined by the number of bits used for representing the integers. The type of these inputs would be simply 'bit'.

The HIFI design methodology, as can be seen from Figure 4.2, thus allows the designer to explicitly specify the decomposition of a function by means of a (regular) DG. Through the process of partitioning such a space decomposition can be mapped onto a SFG, thus implicitly creating a (more complex) decomposition of the function in the time domain. This way time and space decompositions can be seen as equivalent. They lead to different implementations with regard to hardware requirements, throughput, latency and other factors that affect the cost and performance tradeoff that a designer has to make. One of the goals of HIFI was to bring the designer into a situation where these types of tradeoffs could be easily made and compared. This is achieved by letting the designer specify the implementation of a function in the space domain, i.e. as a DG. The decomposition thus specified will lead to a system that has an optimal performance, when mapped directly to hardware, in the sense that the throughput is maximal and the latency minimal. The cost associated with such a system will be high however. In order to reduce the cost, we allow the designer to project the DG on a SFG, that defines a sequential

implementation of the function(s) computable by the DG. The projection is considerably simpler in case of a regular DG. Moreover, regular DG's are projected on regular SFG's, that can be implemented directly on systolic or wavefront arrays.

A dependence graph provides a good basis for subsequent (automatic) implementation steps, because it gives a precise, complete and easily interpretable definition of a function, without restricting its implementation. In case the DG is a so-called shift-invariant DG [Kung86], we can use projection techniques similar to those discussed in [Rao85], that will map the DG on a SFG that can be efficiently implemented on either wavefront or systolic arrays. Another advantage is that DG's can be specified rather easily using a graphics editor. However, expressing a function, down to the lowest level of detail by means of a DG may lead to extremely detailed DG's. Our methodology should provide an answer by allowing decompositions to be specified hierarchically. The decomposition of a function can be made arbitrarily detailed by allowing the definition of intermediate functions, which can in turn be decomposed. The design system has to keep track of all dependence graphs generated which poses additional demands on the database system used for storing the design data.

4.2 Computational Model

The CSP model of Hoare [Hoar85] will be the basis from which we derive our HIFI model. The CSP model, already introduced in section 3.2.2.1, defines a process by its externally observable behavior, i.e. by the sequence(s) of events (traces) in which it is prepared to engage. Communication between otherwise independent processes is accomplished by concurrent composition of the sub-processes. Conceptually processes can exchange values via so-called channels. Theoretically however, a channel and the values it can transport, define a class of events. When that is appropriate we will use the notations introduced in section 3.2.1.1 in order to elucidate our definitions.

The HIFI model, as developed in this section, is a specialization of the CSP model. It distinguishes between input and output events, similar to CSP,

but in addition assumes that the relationship between the values on the inputs and outputs is given by a *function*. The function can be evaluated once the values of all its input events are known. Similarly, once the values of the input events are known, it may participate in some set of output events. The values of the output events are computed by the function. By placing the HIFI model in the framework of CSP, we can apply the same type of reasoning to HIFI processes as to CSP processes, i.e. we may reason about their properties in terms of their trace-sets. The HIFI model will be more restricted however, e.g. in the sense that it will not allow non-deterministic and sequential composition of processes. The HIFI model is thus clearly less powerful then the more general CSP model. The advantage that we get in return is a simpler model and one that is better tailored towards the types of applications discussed in chapter 2. In addition, it should be possible to compose HIFI processes with the more general CSP processes, opening the power of a CSP process, although at the expense of increased complexity.

4.2.1 Signal Flow Graphs

In this section we will give a formal definition of Signal Flow Graphs (SFG). To do so, we will have to define a model for the nodes and define the communication mechanism between nodes. We will also discuss general criteria for the *correctness* of a SFG and proof that SFG's as defined here form a suitable basis for the hierarchical design methodology briefly outlined in the previous section.

HIFI Nodes

The computational activities in a SFG are captured by means of so-called nodes. A node is the abstraction of a computational device, a processor, that can perform any one from a set of functions. Nodes have so-called input and output ports, that carry values. The functions that a node is capable of performing compute the values at (a, possibly empty, subset of) the output ports, the so-called *active* outputs, given values at (a, possibly empty, subset of) the input ports, the so-called *active* inputs. In addition, these functions will also compute the next state of the node. The *state* of a node determines the actual function that it performs, from the collection of

possible functions. The input and output ports of a node, combined with information regarding the types of the values that these ports can handle, define the set of events in which the node is in principle capable of engaging, i.e. its alphabet.

Definition: 4.1

An AST-node is the abstraction of a processor and is defined by:

- a set of inputs, I
- a set of outputs, O
- a set of function-bindings, F
- a state S.

The function-bindings $F = \{fb_1, fb_2, ..., fb_n\}$, relate the inputs and outputs of the node with the inputs and outputs of a function, as will be explained in more detail in section 7.1. The function-bindings F implicitly determine the set of functions that the node is capable of performing. The state S is used to select one of the function-bindings fb_i.

An operational definition of a node can now be given in the form of a CSP process that defines the behavior of the node from the point of view of its environment. Let activeIn(S) and activeOut(S) be processes that consist of input, respectively output events, one event corresponding to each active input respectively output of the node. The events may occur in any order and consist of pairs of port names and values, similar to the input/output events defined by Hoare [Hoar85]. Also, let the notation $\xrightarrow[S^+]{F(S)}$, denote a transition during which the function, say f, associated with the function-binding F(S), selected by S is evaluated. The next state, which is computed by f as well, is assigned to the variable S^+. Now consider the definition given below, where the state of the node is represented by the variable *S*:

$$AST = (S \leftarrow state_0) \rightarrow \mu X.(activeIn(S) \xrightarrow[S^+]{F(S)} activeOut(S) \rightarrow (S \leftarrow S^+) \rightarrow X)$$

The CSP process defined above is given in a recursive form in order to

express the fact that the behavior exhibited by a so-called AST node is cyclic. The behavior of an AST node, according to the equation given above, can be informally stated as follows:

1. Select the active inputs, according to the current state and wait for an event to occur on each one of them.
2. Extract the values from the input events and compute the values of the output events using the function selected by the state. In addition to computing the values of the output events, the function selected by the state is used to compute S^+, the next state.
3. Wait for the output events, selected according to the current state, to occur.
4. Replace the state of the node by the newly computed one.
5. Repeat steps 1-4 indefinitely.

The initial state of the AST node has to be defined; above it is assumed to be $state_0$.

HIFI communication

Given a set of HIFI nodes we wish to combine them so as to create a more complex process †.

In the HIFI system composition is specified by means of SFG's.

Definition: 4.2

A SFG is a directed graph consisting of AST nodes and (directed) edges. The **inputs** of the SFG are the node inputs that are not connected by edges to other nodes. Similarly, the **outputs** of the SFG are the node outputs that are not connected by edges to other nodes.

† Note that in the HIFI system, we actually assume a top-down strategy, i.e. starting from a high-level definition, we decompose it until we arrive at definitions that are sufficiently simple to be implemented by available hardware modules.

Communication between nodes takes place when two nodes that are connected by an edge, representing a channel, are prepared to engage in an input and an output event on that channel.

Usage:

In order to simplify the following discussion, we will in the sequel use the following terminology. We will say that an edge contains a token, if the node that has one of its output ports connected to the edge is prepared to engage in an output event. Also, an edge connected to either an active input port, an active output port, or both, is called an active edge.

4.2.2 *Correctness of HIFI SFG's*

The SFG model discussed above requires verification of the data-flow, because nodes are in no way synchronous and mismatches between the supply and demand of tokens at input and output ports 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 the communication between the nodes in a SFG can be obviously incorrect:

1. all nodes have selected at least one input that is connected to an edge that does not contain a token.
2. there exists a cycle of edges connecting active inputs and outputs such that all edges carry a token.

ad. 1 : In this situation none of the nodes will be able to continue its operation; therefore the SFG is clearly incorrect. Note that the initial supply of tokens comes from either the outside world, or from the nodes that are able to operate autonomously due to the fact that they don't have active inputs.

ad. 2 : In order for communication to take place between two nodes, one of the nodes must be prepared to engage in an input event. In case there is a cycle of edges, all carrying tokens, the nodes connected to the edges are all prepared for an output event. The AST model of the nodes does not allow interleaving of input and output events. Therefore, none of the nodes in the

cycle can perform an input event. As a result the nodes can not continue their operation; they are waiting for each other and again the SFG will be incorrect.

Definition: 4.3

A SFG has a *misfit* if there occurs a situation in which all nodes have selected at least one input connected to an edge that is devoid of a token.

Definition: 4.4

A SFG has a *deadlock* if it has a loop of edges connecting active inputs and outputs, such that all edges carry tokens.

A further danger that threatens the correct operation of a SFG is the possibility of livelock.

Definition: 4.5

A SFG has a *livelock* if its nodes can engage in an infinite sequence of internal communications, without requiring external communication.

Livelock may easily occur whenever the SFG contains a directed cycle, such as the one shown in Figure 4.3. The nodes A and B in Figure 4.3 can spend all their time just exchanging data values between themselves along the edges e_1 and e_2, without ever needing input or creating output.

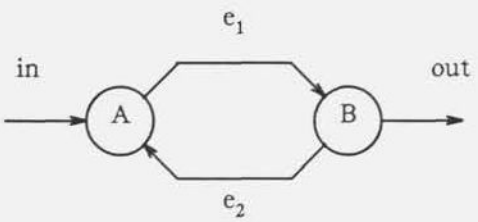

Figure 4.3. Livelock in a SFG

Clearly the system defined by such a SFG is not very useful as a building-block, since it does not communicate with its environment.

We now reach our main theorem, upon which we base our hierarchical design method, i.e. the theorem stating that a SFG consisting of AST nodes can itself be considered an AST node, in the sense that they are I/O

equivalent. In order to state the theorem, we will first define what we will call a 'correct' SFG.

Definition: 4.6

A SFG is *correct* if under no circumstances misfits, deadlocks and/or livelocks are possible.

Theorem: 4.1

For every correct SFG there exists an AST node, such that they compute the same relationship between values on active input and output ports.

Proof:

In order to proof the theorem, we show that the SFG operates according to the AST model defined in section 4.2.1, i.e. a SFG must have a state that determines the function it is to perform in order to compute a pair consisting of the next state and output. The state of a SFG is finite and can be taken as the product of the states of the nodes in the SFG, due to the fact that only nodes have a state. The proof proceeds by showing that there exists an infinite sequence of node operations, due to the fact that there can be no misfits or deadlocks and that, due to the absence of livelock, some of these node operations require interaction with the environment of the SFG, i.e. either the SFG needs input tokens, or it creates output tokens.
Consider first initial functions in all the nodes and consider the graph formed by the edges connecting active inputs and outputs. This graph will be acyclic due to the absence of deadlock. Also, due to the absence of misfits, there will be at least one node that will have tokens on all its active input ports. This node can operate, i.e. it can input the values from its active input ports and output the values it computes on its active output ports. As a result, the state of the node will change and a new graph of active edges will exist. Another node will be able to operate and so on. Therefore, if a SFG does not contains misfits or deadlocks, we will be able to construct an infinite sequence of these node operations, provided of course that if the SFG wants to input a token from the environment, the token will be supplied. Similarly, tokens created on output ports of the SFG have to be taken away.

Depending on the connections of the active inputs and outputs of the nodes with the external environment, the node operations can be classified as follows:

— *input-operations*, if one or more of the active inputs of the node are not connected to other nodes,

— *output-operations*, if one or more of the active outputs of the node are not connected to other nodes,

— *input/output-operations*, if one or more of the active inputs and one or more of the active outputs of the node are not connected to other nodes,

— *cyclic-operations*, otherwise, i.e. if all active inputs and outputs of a node are connected to inputs/outputs of other nodes in the SFG.

We can now identify a global state and global state-transitions and the corresponding global functions of the SFG. First consider all possible sequences of node operations as defined above. These sequences can always be split in subsequences that consist of a (finite, possibly empty) sequence of cyclic operations, followed by an input, an output or an input-output operation. Note that since the SFG does not have a livelock, there can be no infinite sequence of cyclic operations. The global state-transitions of the SFG can now be taken as the combined effect of the node state-transitions corresponding to the subsequences. Similarly, the (global) functions corresponding with the global state-transitions can be constructed by considering the composition of the functions corresponding to the state-transitions of the nodes. Therefore, since the global state-transitions of the SFG are similar to the state-transitions of an AST node and each such state-transition corresponds to a function that involves at least one input, output, or input-output operation, we have proved that a SFG can be represented by an AST node.

□

It follows from the proof above, that the AST node corresponding to a SFG is not unique, unless we also specify how to combine the node operations into global state-transitions. Above we have used the rule that every input,

output, or input-output operation corresponds to such a global state-transition. This was done so as to reduce the number of cases to be considered. For example, an equally valid choice would have been to combine a, possibly empty, sequence of input transitions with a, possibly empty, sequence of cyclic transitions followed by a, possibly empty, sequence of output transitions. However, in order to be complete, we would also have to consider all possible interleavings of the cyclic transitions, with a single input-output transition. For the purpose of the proof above, this construction is clearly not needed.

In order to determine the correctness of a SFG, it will be necessary to know the traces of the AST nodes that it contains. The trace-set of an AST node can be computed if we can determine its state transitions. In general, this will be very difficult, if not impossible, due to the fact that the next state of a node may depend on the values of all input events. These values are determined by the environment of the node. Therefore, unless the state-transitions are fixed and thus become independent of the values of the input events, we can not determine the trace-set of an AST node a priori. This of course is very undesirable, since many important properties of a system, e.g. the absence of deadlock, can only be determined if we know the traces of the processes used in constructing the system.

An alternative way of establishing the correctness of a SFG is by simulating it. Although simulation can not be used to determine correctness with absolute certainty, it can be used to verify the correctness for certain sets of inputs. Using a simulator one can also determine the *functional* correctness of a SFG, i.e. whether the relationship between input and output values is what the designer intended it to be.

A simulator that is particularly suited to the simulation of SFG's was developed by Held [Held87]. This simulator allows the definition of nodes by means of ordinary subroutines, written in C or a similar programming language. The simulator is able to invoke a large number of such subroutines in parallel, using a co-routine like mechanism. The communication between nodes is done by means of special functions that can be called from within the subroutines defining the nodes, as any

ordinary function can be called. These I/O functions implement a single token pass protocol that emulates the communication along an edge in a SFG. In order for communication to take place, two nodes must simultaneously call input and output functions, called get and put respectively. If the calls do not take place simultaneously, then one of the functions, the one that was called first, will enter a wait state. This wait state allows the function to wait for a matching input or output action.

The scheduler that implements the co-routine mechanism, can detect both deadlock's and misfit's. The scheduler can then give control to the user, who can inspect the SFG, in order to determine what caused the deadlock or misfit. Detection of livelock is based on interaction between the simulator and the user, e.g. by displaying a view of the activity taking place in the SFG. The user is able to interrupt the simulator whenever he suspects livelock, for example in case a token starts circulating in the SFG. Upon interruption, the states of the nodes can be inspected.

4.2.3 DCM and functional SFG's

In this section we will discuss two specializations of the more general AST nodes defined in section 4.2.1, respectively FNC nodes and DCM nodes. These nodes are introduced here so as to allow the correctness of SFG's to be more easily determined (a priori). DCM and FNC nodes do not have a state-transition mechanism; therefore, they do not have a state of their own. As a result, in order to be able to specify systems that do have a state, it will be necessary to introduce two special nodes, respectively *delay* and *buffer* nodes, that have a known, and in fact very simple trace.

If we let c?x denote an input event, that assigns the value read from channel c to the variable x and likewise let c!x denote an output event that writes the value of the variable x to the channel c, than we can define a delay node as follows:

$$\text{DELAY} = \text{state} \leftarrow d_0 \rightarrow \mu X.(\text{out!state} \rightarrow \text{in?state} \rightarrow X)$$

Similarly, a buffer node is defined as:

$$BUFFER = \mu X.(in?state \rightarrow out!state \rightarrow X)$$

Since delay and buffer processes have a single input and output, we can easily denote them by adding a weight to a SFG edge. This weight is specified in the form of D's or B's, as was already shown in Figure 4.1(b). The delay node shown there has the same trace as those defined above, which serves to show that a delay can be defined as an AST node. The initial value of the state of a delay is a parameter of the delay process, which has to be specified. In case it is left unspecified, we assume that it has a suitable 'null' value.

4.2.3.1 Functional SFG's

A FNC or function node does not have a state-transition mechanism, due to the fact that it can perform only one function. The definition of a CSP process implementing the mechanism associated with a FNC node is therefore considerably simpler then that of an AST node.

$$FNC = \mu X.(inputEvents \xrightarrow{f} outputEvents \rightarrow X)$$

In this case, 'inputEvents' is a process used to input the values on the input ports in an arbitrary order. Similarly, 'outputEvents' is a process used to output the values computed by the function f on the output ports in an arbitrary order. There is no selection of active ports in this case; all ports are active all of the time.

4.2.3.2 DCM SFG's

The second specialization of an AST node is a so-called DCM or Distributed Control Model node. A DCM node has a function f_c that is used to compute the next state. As was the case for AST nodes, the state is used to select a function that maps the (values of) events on a subset of the input ports, the so-called active inputs, to the values of the events on a subset of the output ports, the so-called active outputs. The control function f_c uses the values from a distinguished set of input ports, the so-called control inputs. In addition to computing the state of the node, the control function may compute output values. The values computed by f_c are outputted via the so-called control outputs.

A CSP process implementing the state-transition mechanism of a DCM node can be defined as follows, where controlIn and controlOut represent processes that can input, respectively output the control tokens in any order:

$$DCM = \mu X.(controlIn \overset{f_c}{\underset{S}{\gg}} controlOut \cup activeIn(S) \overset{F(S)}{\rightarrow} activeOut(S) \rightarrow X)$$

The advantage of the DCM model over the more general AST model, is that we need to know only the values at a subset of the input ports, the control inputs, in order to determine the state-transitions and thus the trace-set, of the DCM node. The separation of control and data inputs (and outputs) appears to be very useful. In fact it resembles the way conventional digital systems, e.g. microprocessors, are structured. Such systems can be decomposed in a controller and a data-path. The controller consists of a control memory and, typically, a simple FSM to cycle through a sequence of control states. The controller may be represented by a control function that, based on the values stored in the control memory, computes the setting of the data-path, i.e. it computes the function to be performed by the data-path. The values stored in the control memory have been specified somewhere in the design of the system.

The HIFI system will formalize this in the sense that different functions may be decomposed using the same Dependence Graph. If the nodes in this graph are of the DCM variant defined above, then the designer can, by specifying values for the control events, select the function that will be computed by the DG.

4.2.3.3 Correctness

Based on our definitions of FNC and DCM nodes, as well as delay and buffer nodes, we can now investigate the conditions under which SFG's build using these nodes are correct. We will first consider SFG's that are build entirely from FNC and delay nodes. Such SFG's are actually identical with the SFG's used by Kung [Kung86], and many other researchers; we will call them *functional* SFG's.

Property 4.1

A functional SFG is correct if:

- every connected component has at least one input or output and
- every cycle contains at least one delay node.

Proof: This property can be easily verified by checking the conditions that guarantee that a SFG is correct, i.e. absence of deadlock, misfit's and livelock's. First misfits. Since a FNC node does not select active inputs and/or outputs, a functional SFG can not contain misfits. Absence of deadlock is equally easy. The first observation is that, if the SFG is acyclic, then it can not contain a cycle of active edges. The second is that if the SFG would contain cycles, then every cycle contains a delay node. Since the delay allows its output to be active before its input, there is no deadlock. Finally, the absence of livelock is guaranteed by the fact that the nodes in a connected component must operate equally often.

□

The next step is to consider SFG's that contain DCM nodes. In order to compute the traces of a DCM node, we require that the values on its control inputs are known, or can be computed a priori. We will further assume that every control input repeatedly inputs the same sequence of control tokens. These control sequences can be different for different control inputs. The length of all control sequences will be the same however, say n, due to the fact that the control graph is a functional SFG and the behavior of the SFG must be cyclic.

The computability of a SFG with DCM nodes can be determined in two steps. In the first step, we determine the computability of the so-called control-graph, i.e. the graph that remains if all edges not connecting control inputs and outputs are left out. The control graph is a functional SFG, therefore it is computable if every directed cycle contains at least one delay node. The control functions of the DCM nodes compute the states of these nodes. The state of a DCM node determines its active inputs and outputs. The active inputs and outputs of all DCM nodes with the edges that connect

them form together a so-called data-graph (see Figure 4.4).

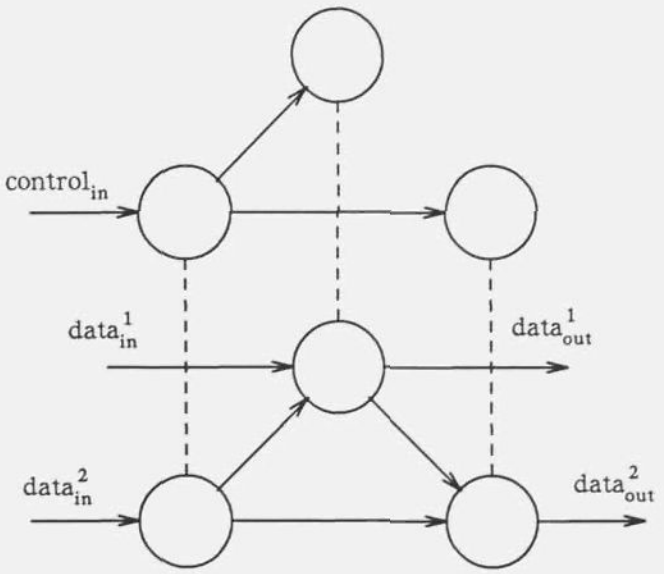

Figure 4.4. DCM SFG: Control (upper part) and data-graph (lower part)

For every possible combination of control tokens on the control inputs, we can thus find a data-graph. Since we assume that every control input repeatedly inputs a value from a sequence of n control tokens, we thus find a sequence of n such data-graphs. A data-graph is again a functional SFG; for it to be correct we require that every directed cycle contains at least one delay node. The determination of the correctness of the SFG as such is complicated by the fact that some edges in the data-flow graph are connected only to an active input or output and not to both. If an edge is connected to an active input, we require that the edge contains a delay node; if the edge is connected to an active output, we require that it contains a buffer node. These buffer and delay nodes can supply or absorb one token. However, by doing so they are effectively changed, i.e. a buffer becomes a delay and a delay becomes a buffer. This will change the weight of the edge which may effect the correctness of the subsequent data-graphs.

Property 4.2

A DCM SFG is correct when its control-graph is correct and for every possible combination of sequences of control tokens on its control inputs, the corresponding data-graphs are correct†. In addition, the weights of the edges and the states of the delays must be the same before and after all operations corresponding to a sequence of control tokens on each of the control inputs of the SFG.

Proof: A DCM SFG under the conditions stated above will not contain deadlocks since the control and data-graphs are required to be correct. Misfits could occur due to the fact that a node may output a token, which is not immediately removed by another node, or if a node needs a token while another node is not yet prepared to send it. These misfits can be removed however with the help of buffer and delay nodes. By verifying the correctness of a sequence of data-flow graphs, considering the (temporary) transition of buffer into delay nodes and vice-versa, it can be checked that there are no such misfits. Other misfits can not occur, again due to the fact that the control and data-flow graphs are required to be correct. Finally, the absence of livelocks is obvious from the fact that the the control graph is a functional SFG.

□

A procedure to verify the correctness of a SFG with DCM nodes, as discussed above, can be easily defined. It suffices to check that the active inputs and outputs of each node in each data-graph are correctly connected, i.e. either to active inputs or outputs of other nodes, to external inputs or outputs, or to delayed, in the case of an active input, or buffered edges, in the case of an active output.

---

† In fact the requirement of correctness can be relaxed, since it is not necessary that both the control graph and the data-graphs contain external inputs or outputs.

4.2.3.4 Dependence Graphs

Definition: 4.7

A Dependence Graph is an acyclic SFG that contains only DCM and/or FNC nodes.

A DG that consists of only FNC nodes, will be called a *functional* DG. The correctness of a DG can be easily verified, due to the fact that it does not contain cycles and that there are no delay and/or buffer nodes. In fact the correctness of a DG is directly related with the values of the control tokens on its control inputs. If the data-graph corresponding to the control tokens does not contain misfits, then the DG is computable. In the sequel we will use these DG's to define decompositions of functions. A functional DG defines the decomposition of a particular function; otherwise, a DG can be viewed as defining the decomposition of a collection of functions, one corresponding to each combination of control tokens on the control inputs.

4.2.4 Examples

In order to show the expressiveness of the model we consider LIFO and FIFO buffers. A LIFO buffer of depth 4 is shown in Figure 4.5.

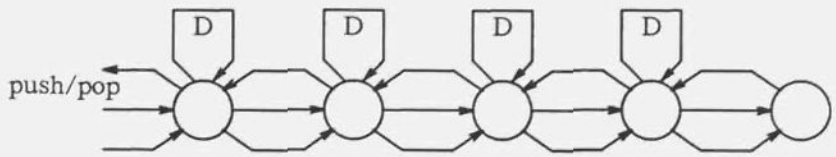

Figure 4.5. LIFO buffer of depth 4

Each node can perform one of two functions, called 'push' and 'pop'. The push function reads a new value in the LIFO. The values in the LIFO are stored in the delay edges. When pushing a new value in the LIFO, all nodes eventually perform a push function. This propagates the values in the delay edges to the right (cf. Figure 4.5). The rightmost node serves as a source or sink of tokens (data-values), depending on whether it is requested to pop or push a token. The control functions of the node simply propagate the control tokens. The correctness of the SFG shown in Figure 4.5 can be verified by considering the data-graphs associated with the 'pop' and 'push'

control tokens, shown in Figure 4.6.

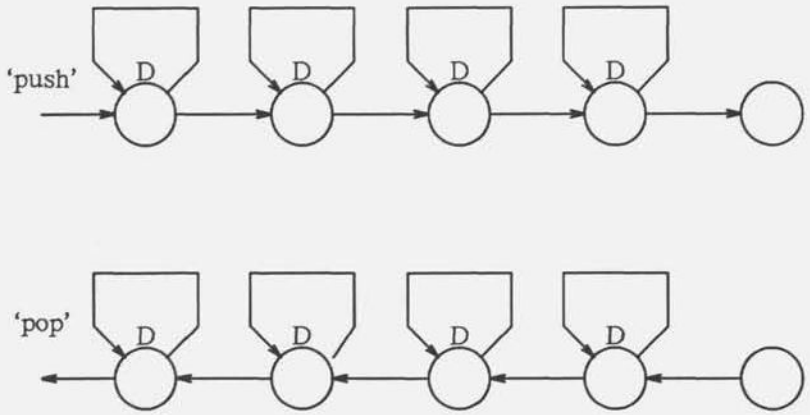

Figure 4.6. data-graphs of the LIFO

Both data-graphs are correct, i.e. they contain no misfits or deadlocks. In addition, the weight of the edges is not changed. As a result we can conclude that we can combine the two data-graphs in an arbitrary order. The correctness of the SFG is guaranteed no matter what sequence of control tokens is put on the control input. The problem with the LIFO buffer as defined by Figure 4.5 is that, in order to pop a value, it is necessary to propagate all values from right to left. The control token has to travel through the nodes from left to right and only after it has arrived at the rightmost node, can the data value start propagating. Such a system can't be implemented on a systolic array, although the SFG itself is correct and can be implemented on a wavefront array.

The second example is the FIFO buffer shown in Figure 4.7.

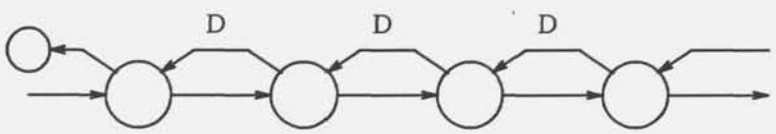

Figure 4.7. FIFO buffer of depth 4

Each node has a control and data input and output. The edges connecting the control inputs and outputs are delayed. The operation of the FIFO buffer depends on the availability of tokens on its inputs. For each token on the control input, it will deliver a token on the data output, subject to the availability of tokens on the data input. The control tokens coming from the leftmost node have to be removed, e.g. by adding a node that simply accepts tokens on its input. Due to the delays on the control edges, the nodes can initially shift in a number of values equal to the depth of the buffer.

5. HIFI: Function Decomposition and Implementation

5.1 Refinement: Function Decomposition

As explained in section 4.1, the principal means of abstraction in the HIFI system is a function, viewed as a mapping from inputs to outputs. In this section we will discuss the methods provided in the HIFI system to decompose functions. A function has to be decomposed, in order to arrive at an implementation. Associated with the decomposition of the functions we will find that we have to decompose the values at their inputs and outputs.

Usage:

A decomposition of a function in the space domain (cf. section 4.1) will be called a *refinement*.

The HIFI system distinguishes two methods of refinement:

- structural
- regular

5.1.1 Structural Refinement

Structural refinement can be thought of as the replacement of a function by a functional Dependence Graph. The nodes in such a DG perform a single function. A structural refinement, i.e. a DG, is most easily specified by means of a graphical editor. It can however also be specified as a set of equations relating the inputs and outputs of the nodes (functions) in the DG, or as a functional program (cf. section 5.1.3). In addition to the structure of the DG a designer will also have to specify the binding of the inputs and outputs of the DG to the inputs and outputs of the function being refined, say F. The binding may involve a decomposition of the data-types associated with the inputs or outputs of F. In general a composite type, i.e. a type representing a tuple of values, may be decomposed in its value types. Data-type decomposition is specified as part of the binding of a DG input or output to a function input or output.

An example of a structural decomposition is given in Figure 5.1. The function *floating multiply* is decomposed using three functions, that respectively add the exponents, multiply the mantissa's and perform a normalization of the resulting floating point number.

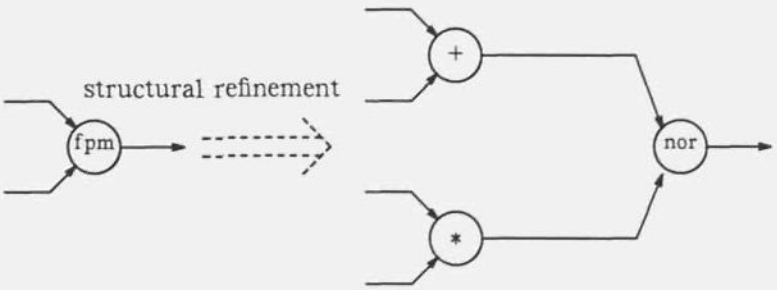

Figure 5.1. Decomposition of a floating point multiply function

The example also shows the decomposition of data-types as it is necessary to decompose the type of the inputs of the function *fpm* in an exponent and mantissa type.

5.1.2 Regular Refinement

A regular refinement of a function is used when a function can be implemented by repeated application of another function, the so-called iteration function. Functions that can be decomposed in a regular fashion can be efficiently implemented on systolic arrays and are thus of considerable interest in the context of the HIFI system.

A regular refinement of a function is specified by defining an index space and a set of directed edges connecting the grid points belonging to the index space.

Definition:

An index space is a lattice enclosed in a region of the n-dimensional Euclidean space, defined by a set of constraints such that a point I belongs to the index space only if it is a feasible solution to the set of constraints.

The edges connecting the grid points of an index space are regular, i.e. if one grid point I is connected to a grid-point J = I + D, then all grid-points I are connected to grid-points I + D, provided both I and I + D belong to the index space. If the point I + D does not belong to the index space, then I is an output point of the index space. Similarly, if I - D does not belong to the index space, then I is an input point of the index space. The iteration function defines a node. Similar nodes are positioned at all point belonging to the index space. The dependencies specify the connections between the inputs and outputs of these nodes. The inputs of the nodes that are associated with the input points of the index space are bound to the inputs of the function being refined. Depending on the dimension of the index space and the position and number of inputs, this requires an appropriate decomposition of the data-types associated with the function inputs.

An example of a regular decomposition is given in Figure 5.2. There we show a regular decomposition of the function *integer multiply*.

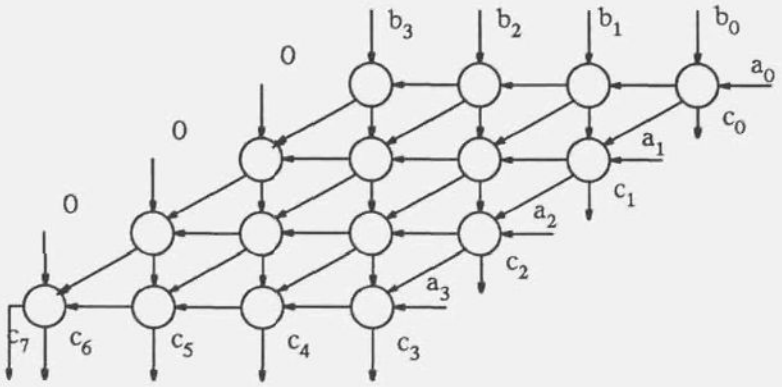

Figure 5.2. Regular decomposition of integer multiply

The integers a and b at the inputs of the function are decomposed into their bit representation. The number of points in the index space depends on the precision, i.e. the number of bits, used for representing the input values a and b.

The constraints used for defining the index space can be defined in a number of ways. A straightforward method is to associate an interval and stepsize with each dimension of the index space. In most cases, the number of points in such an interval will correspond to the number of data elements into which a value on a function input or output gets decomposed. As a result, the size (i.e. the number of points in the index space) will usually depend on an attribute of a type associated with an input or output of a function that is decomposed.

In order to define a regular decomposition of a function we need to define:

- the dimension of the index space.
- constraints to identify the points belonging to the index space.
- an iteration function, or DCM node (see below).
- a set of dependencies.
- input and output bindings, i.e. type refinements.

5.1.2.1 DCM Nodes

The domain of application of regular refinements is greatly increased if, instead of using FNC nodes in the Dependence Graph, we use DCM nodes. First of all, by defining an appropriate control function f_c, it is not longer necessary that all nodes perform the same function. For example, nodes at the boundary of the index space may perform a slightly different function in order to ensure a proper initialization. The function performed by the nodes, is selected depending on the value of the control tokens on the control inputs of the DG. In order to allow regular refinements to use DCM nodes, the designer will have to define a DCM node and in addition specify an appropriate set of (external) control tokens, such that the refinement indeed computes the same I/O relationship as the function being refined.

5.1.3 Functional DG specified by a functional program

As was already briefly mentioned, a functional DG can be associated with a functional program and vice-versa, provided that we can define appropriate denotations for the combining forms, i.e. the higher level functions, used in

such programs. Two simple examples will clarify this. The first one regards composition. Composition is a combining form that is used to specify pipelining. The composition of two functions, f_1 and f_2, denoted by $(f_2 of_1)$ can be represented by a DG that has two nodes, where the outputs of the first one, that performs the function f_1, are connected to the inputs of the second one, as shown in Figure 5.3(a). The second example regards construction. Construction is a combining form that is used to specify parallel composition. The construction of two functions, $[f_1,f_2]$, can be represented by a DG as well. The definition of the DG is however somewhat more complicated, due to the fact that we have to duplicate the inputs. This is a consequence of the way in which we have defined the communication mechanism. The resulting DG is shown in Figure 5.3(b).

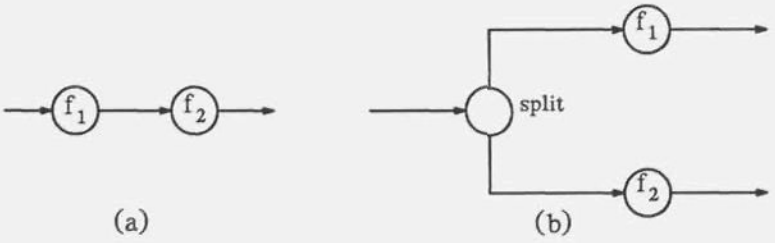

Figure 5.3. Construction represented by a DG

More complicated combining forms can be defined in order to define e.g. regular DG's. Sheeran [Shee83] defines several such forms, e.g. to define regular 1D and 2D arrays. The combining forms that are used to define these regular structures, are usually defined recursively, based on the *form* of the input sequence, i.e. the number of elements in an input tuple.

The definition of combining forms and the derivation of a body of laws regarding their application, can be the basis of a more formal definition of function decomposition. It appears possible to do so [Jone86], although a lot of work will be needed before these concepts can form a formal basis for a design system.

5.1.4 Hierarchical Refinement

In order to simplify the definition of a complicated decomposition, we will require that the design system allows the designer to specify a particular decomposition as a sequence of regular decompositions and/or structural decompositions. For example, to define a regular decomposition of a matrix-matrix multiplication, we may first define a 2-D regular decomposition in which the iteration function computes the inner product of two vectors (cf. Figure 5.4(a)). The inner product function may next be decomposed, using a 1-D regular decomposition, into a linear array of inner-product step processors (cf. Figure 5.4(b)). The inner-step function can then be decomposed in a multiplier and an adder (cf. Figure 5.4(c)).

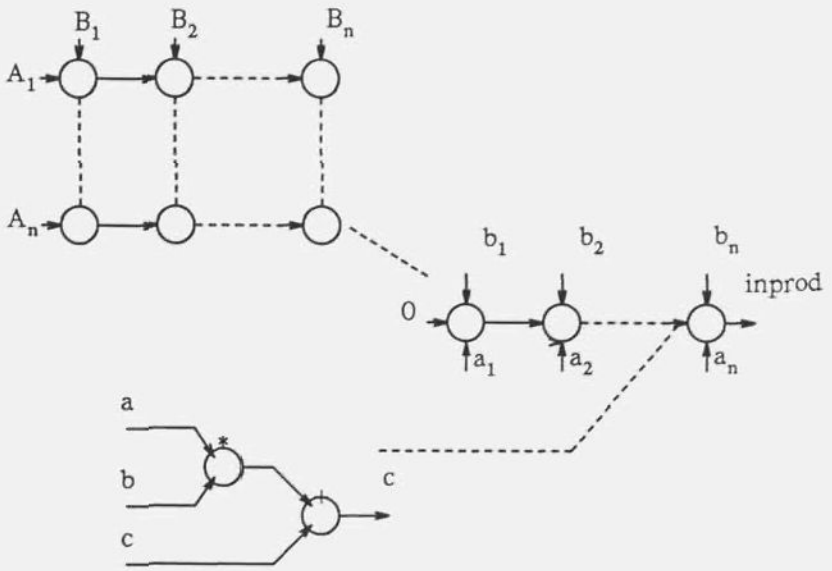

Figure 5.4. Hierarchy of refinements

In order to define an implementation the designer should have the possibility to expand any of these decompositions, by instantiating lower level decompositions.

5.2 Partitioning: Function Implementation

An implementation of a function is created by mapping a Dependence Graph, that specifies a refinement of the function, onto a SFG. The SFG realizes the function by a sequence of state transitions and uses less nodes (processors) at the expense of the memory (delays and buffers) needed to store the state. The function computed by the SFG, when assuming that the SFG maps sequences of inputs to sequences of outputs, is determined by the sequences of control tokens on the control inputs. The procedure used for mapping the DG onto a SFG depends on the properties of the DG, more specifically whether it is regular. A regular DG can be mapped on a SFG by choosing a projection vector U. The nodes in the DG that are on lines parallel with U are mapped on the same SFG node. A DG resulting from a structural decomposition can't be so easily mapped. It will be necessary to partition the DG into classes that can then be implemented by a single SFG node. In the case of regular DG these classes where implicitly determined by the projection vector U. In the case of structural refinement we need a different mechanism. The best method appears to be one that takes into consideration the final (hardware) implementation, optimizing e.g. the trade-off between processor complexity, I/O bandwidth and throughput rate. Such a method may however need detailed information on the available hardware. An alternative is to rely on the designer for specifying the classes. Using a graphical representation, the designer may for example specify the classes, by drawing boundaries along specific sets of nodes.

5.2.1 Regular Partitioning

In case of a regular DG the partitioning procedure is very simple. If we represent the dependencies by the columns of a matrix D, then the procedure consists of the following steps:

- Choose a projection vector U, such that $U^tD \geqslant [0\ 0\ ..\ .\ 0]$.
- Construct a $(n-1)\times n$ projection matrix P, such that $P^tU = 0$, i.e. the basis of the index space of the SFG has to be orthogonal to the projection vector U.

- Determine the extent of the index space of the SFG, e.g. by projecting all index points of the DG on the corresponding index points of the SFG.
- Compute the dependencies D_{sfg} of the SFG: $D_{sfg} = PD_{dg}$.
- Compute the weights of the dependency vectors D_{sfg}: $W_{sfg}=U^tD_{dg}$.
- Project the inputs and outputs of the DG on the input and output sequences of the SFG. This is done in two steps: (1) determine the input/output on which the value associated with a DG input/output will appear and (2) compute the order in which the values will appear on the SFG inputs/outputs. If we let I and J denote the positions of the input/output in the index space of the DG, then the ordering relation required in step two is: $U^tI \leqslant U^tJ$.

The above procedure has to be extended in case the DG contains edges parallel with the projection vector U. Such edges are projected on (SFG) edges whose weight is greater then 0, i.e. edges that have buffers. The first value from such an edge has to come from the environment however. Similarly, the final value has to go to the environment. Two solutions are possible:

1. to provide switches that allow the first, respectively last, value to be read from, respectively written to, the environment.
2. to modify the control of the SFG.

The first solution is shown in Figure 5.5. The 2-D DG is projected on a SFG that has three columns of nodes, respectively a column of input switches, a column of 'f' nodes and a column of output switches. One of the outputs of the output switches is connected to one of the inputs of the input switches, such that the edge forms a feedback edge.

The switches keep track of the number of times the data is circulated through the 'f' nodes. The input switch (IS) initially reads data, i.e. a token, from the left and then circulates it (N-1) times, where N is the number of nodes that are projected on the SFG node. Similarly, the output switch (OS) node circulates the data (N-1) times and then outputs it to the

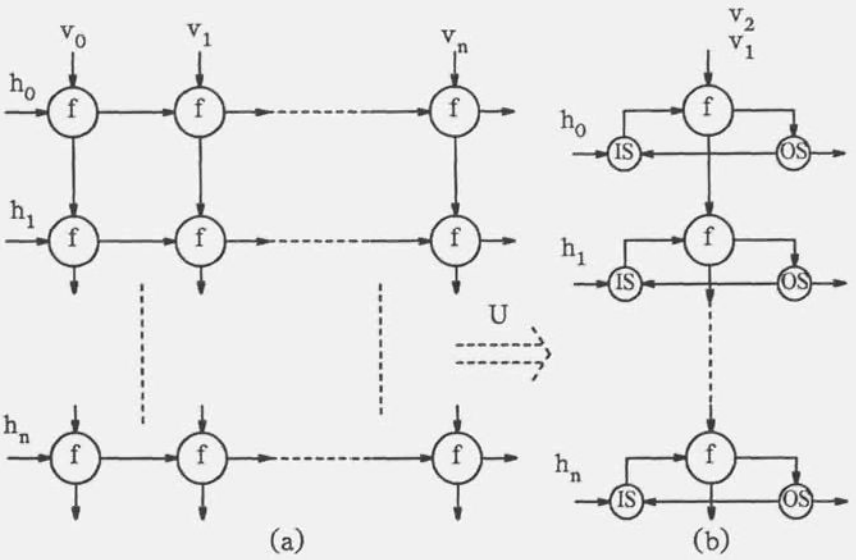

Figure 5.5. Switches used to initialize the feedback edges

right. The input and output switches are DCM nodes, whose control function increments a control index, stored in the delay associated with a (feedback) edge, that connects the control input and output of the switch, as shown in Figure 5.6.

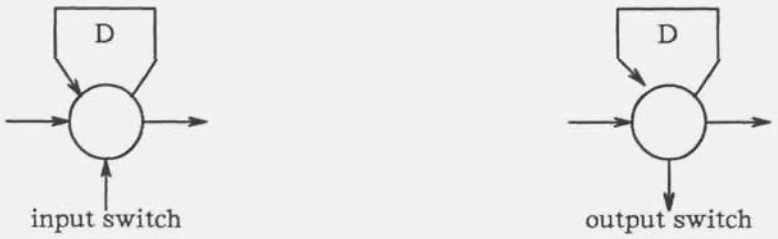

Figure 5.6. Input and output switch

The function performed by a switch depends on the value of the control index. Note that the input and output switch also provide the buffering that is necessary to make the SFG correct.

If we replace each dependency parallel to U, with a feedback loop as shown in Figure 5.5, then the partitioning procedure described above, will generate a correct SFG.

The second solution is shown in Figure 5.7. This solution is however only possible when the DG contains control dependencies that are not parallel with the projection vector U. If that is not the case, then we can modify the control function, such that during the first, respectively final iteration of each cycle, the node reads, respectively writes a token from the (external) input, respectively output. The first and final iteration are distinguished by modifying the associated control tokens, respectively c_0 and c_n.

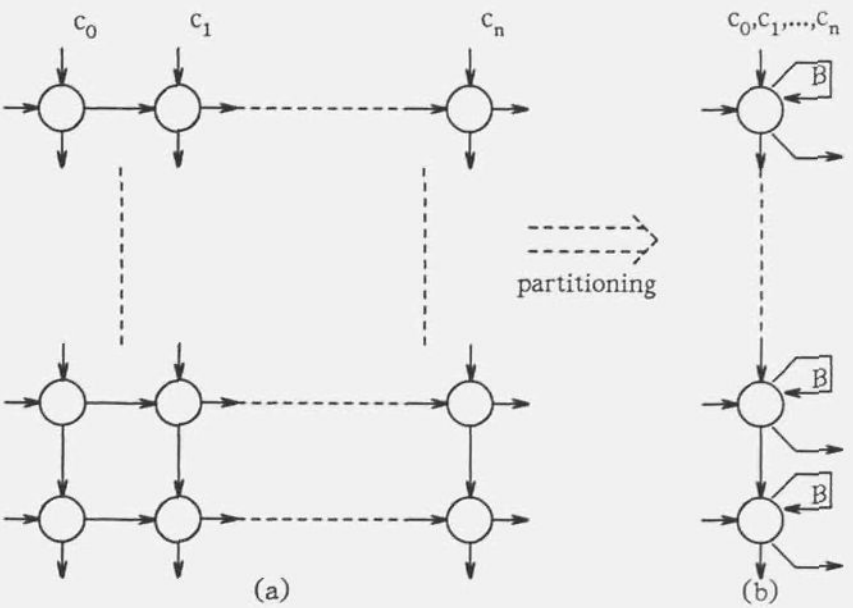

Figure 5.7. Control function used to initialize the feedback edges

The values that come from the left in Figure 5.7(a) are the initial values for the buffers shown in Figure 5.7(b). They are read from the environment by modifying the first control token c_0, such that the corresponding function will take its input from the left, instead of from the buffer. Similarly, the final control token, c_n has to correspond to a function that writes its output

to the right, instead of putting it in the buffer. The functions corresponding to the intermediate tokens c_1 take their input from the buffer, as well as put their output there. Therefore, in order to properly initialize the buffer, it is necessary to distinguish the first and final control token from the intermediate tokens and from each other. Only then can the SFG node select the appropriate function.

5.2.2 Structural Partitioning (Clustering)

In case of a structural refinement, the mapping of the nodes of the DG on the nodes of the SFG can be achieved by having the designer specify the nodes that are to be implemented by a SFG node. For each cluster of DG nodes, the system will have to generate a sequence of control tokens, such that the SFG node computes the same values computed by the nodes of the DG. Values that are intermediate, i.e. values that are used only inside the cluster, contribute to the state, in the form of (buffered) feedback edges, of the SFG. The edges connecting nodes in the cluster with nodes outside, become the inputs and outputs of the SFG node. One method to add the control is to let every SFG node have a control input and control output, connected by a feedback edge. The number of delays on this edge corresponds to the number of nodes in the cluster. With the initial values of the delays in fact a simple control program is implicitly specified, that lets the SFG node execute a sequence of functions. The data-flow communication between the nodes ensures a correct synchronization of these local programs.

A simple example is shown in Figure 5.8. There we map a cluster of three (DG) nodes, on a SFG node that can implement all three functions. The SFG nodes are controlled by means of the values stored in the delays. In the example shown, the SFG nodes will repeatedly perform the functions f_1, f_2 and f_3 in sequence. Each cluster contains three internal edges, that are projected on edges that contain a buffer to store the intermediate value until it is needed.

5.2.3 Partitioning of SFG's

The partitioning procedures described above apply only to the partitioning of DG's. In this section we will discuss a more general procedure that also

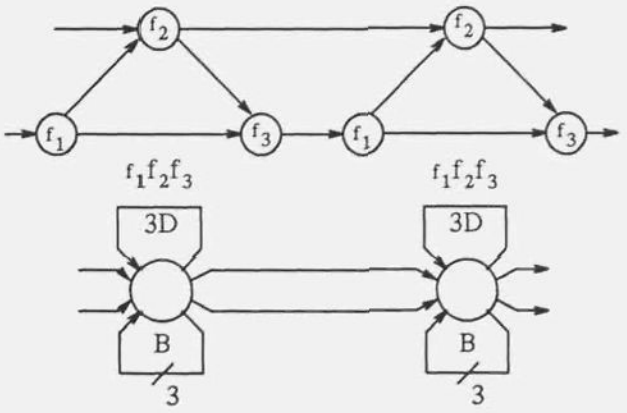

Figure 5.8. Structural Partitioning

allows us to partition regular SFG's, i.e. SFG's generated by applying the partitioning procedure to a regular DG.

The simplicity of the partitioning procedure in section 5.2.1. derives largely from the fact that neither of the nodes, nor any of the edges in a DG has a state. This is because the nodes are FNC or DCM nodes and the edges have no buffers and/or delays associated with them. A (regular) partitioning of such a DG will in general result in a SFG that has buffers associated with its edges. However, if the SFG is the result of a partitioning procedure as described above, then after every sequence of control tokens, i.e. after every what we will call *cycle* of the SFG, the buffers will be empty, i.e. stateless. It follows that if we want to partition a SFG, we should not separate the tokens belonging to an input sequence.

To implement this we need to have slightly more complex input and output switches. To define the new input and output switches, we need to introduce a second parameter, which we will call the sequence index, in order to distinguish it from the other parameter, which we will call the node index from now on†. The sequence index is used to keep track of the

position of a (data or control) value in a sequence. The control function of an input or output switch will now decrement the node index only once after every sequence index iterations. In addition, due to the fact that every node now has to process a sequence of values, before starting a new (node) iteration, the feedback edge connecting the input and output switches, has to have sequence index - 1 buffers associated with it.

The SFG resulting from a further partitioning of the SFG shown in Figure 5.5(b), is shown in Figure 5.9.

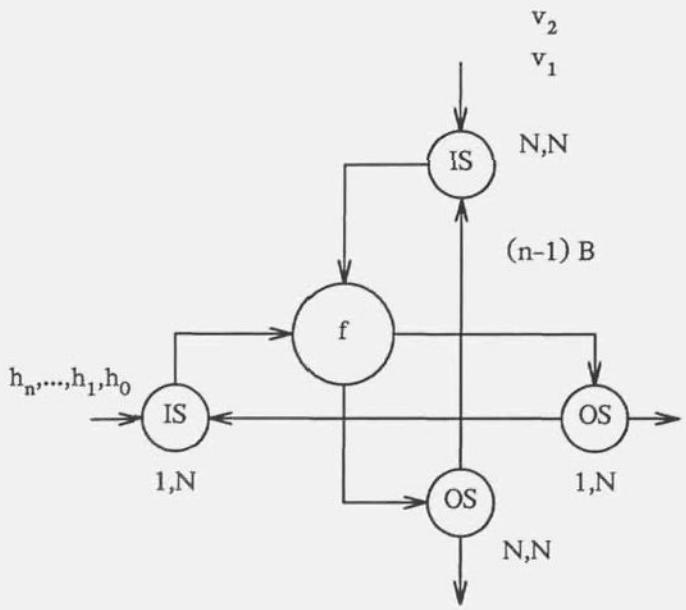

Figure 5.9. Partitioning of a SFG

† The input and output switches defined in section 5.2.1. are special cases of the new switches, for which the value of the sequence index is simply 1.

The partitioning procedure can be further modified so as to allow the number of nodes between the switches to be varied, in order to control the amount of pipelining that takes place. The number of nodes in the projection direction in the partitioned SFG, say p, can be any integer divisor of the number of nodes N in the projection direction in the (unpartitioned) DG or SFG, i.e. N mod p=0. In order to account for this change, we can simply modify the node index of the switches to N/p instead of N, as shown in Figure 5.10 for the case that in the vertical projection p = 2.

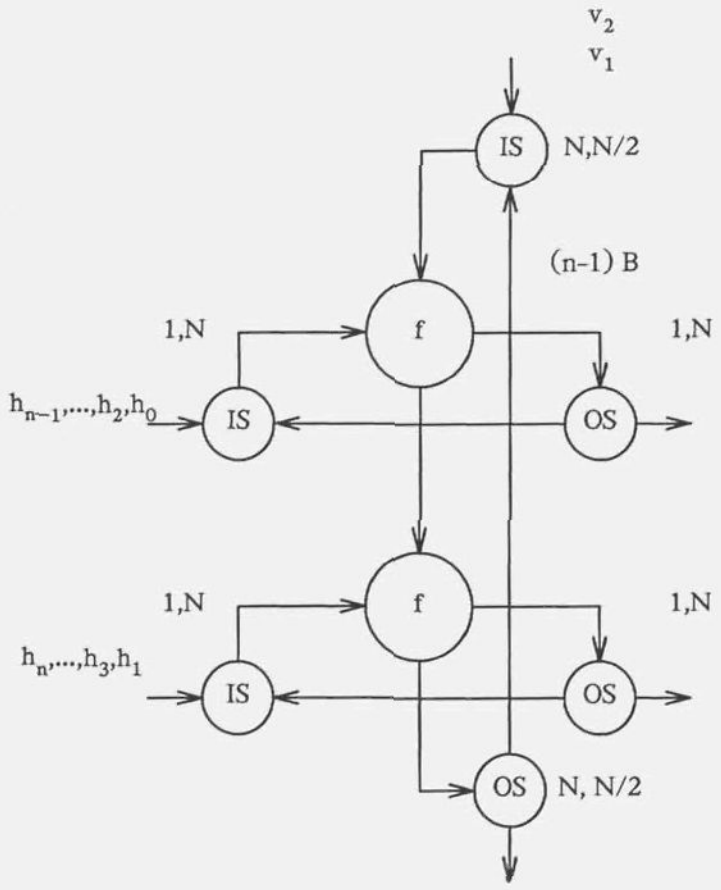

Figure 5.10. Varying the amount of pipelining in a SFG

6. HIFI: Prototype System

The prototype system described in this thesis is of limited scope; its primary purposes are to (1) show, by means of example, the style of interaction between designer and design system and (2) to provide a vehicle for stating and formulating requirements regarding the design system, the way it is setup and the (hardware and software) requirements on the environment in which it is to function.

One of the main complications to be dealt with when setting up a design system, is that a design language like the ones discussed in chapter 3, is not a suitable tool for design; it can only serve to document a design once it reaches a certain state. Furthermore, designing is not a linear activity. Designers frequently backtrack and there are many things that can be left unspecified initially, but need to be filled in later on. This requires the design system to be very flexible regarding the consistency of the design. The designer must be able to control the application of consistency checks. On the other hand the design system has to enforce consistency checks when that is needed to ensure overall consistency. This will allow the definition of a system that allows a truly hierarchical style of design combined with stepwise refinement of specifications. Such a system however poses stringent demands on the environment (hardware and software) in which it is to operate.

Following the object oriented approach pioneered by the developers of SMALLTALK [Gold83], a design, like almost anything else can be defined as a collection of objects which model design entities and relationships between these entities. In order to define a design system, what we have to do is to define object types to model the design entities and their relationships. Examples of design entities in the HIFI system are easily found: functions, refinements, data types etc. The definition of the object types requires definition of their functionality, i.e. one has to define the operations that can be performed by the objects. A particular design activity can then be decomposed in a sequence of object operations. A designer can specify a design by invoking operations on existing or newly

created objects.

The flexibility of the design system is largely determined by the mechanism available to identify design objects and to apply operations to them. In the SMALLTALK environment, this is provided for by the elements of the Model View Controller (MVC) model.

Model View Controller model

> A design entity is modeled by an object, referred to as the model. Models are accessible to a designer via views. A view is a 2D representation of the object on the display screen. A model can have any number of views associated with it. Each view has its own controller. The controller makes it possible to invoke operations on the design object shown in the view.

The flexibility of this model is derived in large part from the fact that the designer can choose the currently active controller by moving the mouse over the display screen. The view that contains the mouse is the currently active view; its controller the active controller. This behavior may be modified, but the principle of selecting an active view/controller pair by moving the mouse over the display screen remains.

The SMALLTALK programming environment supports the definition of design entities via classes. A class defines a set of objects that respond to the same set of messages in the same way. To define a class one needs to define a representation, i.e. a set of instance variables and a set of messages to which instances of the class should respond. In addition, for each message, one has to define a method that is to be invoked when an instance of the class receives that message. The method may change the value of instance variables and/or compute a value based on the values of the instance variables. Since SMALLTALK knows only about objects, the value of an instance variable is a pointer to an object. This is true even for objects like integers and character strings. Consequently, all computing is done by sending messages to objects.

SMALLTALK allows the definition of classes using specialization. A class may be a subclass of another class, in which case it inherits the behavior of

that class, i.e. its message set and representation. In addition, one can define additional instance variables, thus extending its representation and define new messages and associated methods. One can also redefine existing methods. Specialization is a major tool in setting up SMALLTALK applications. The SMALLTALK system provides almost all of the functionality for setting up a user interface based on the MVC model discussed above. Creating a SMALLTALK application consists of defining a model object and constructing views and associated controllers for manipulating and inspecting it. This removes much of the complexity usually involved with developing an application and allows one to concentrate on the essential concepts, i.e. the definition of (one or more) models to represent the design entities.

Although SMALLTALK probably still is the best developed object-oriented programming environment, it is not the only one. Advanced object-oriented programming systems have also been developed to run on LISP machines. Another interesting development are so-called hybrid languages. For example, Objective C [Cox86] allows the development of powerful program development environments that in some way combine the best of two worlds, i.e. the efficiency of a procedural language and the flexibility of an object oriented language offering run-time binding, dynamic memory management, type inheritance etc. An example of such an environment is the RMG system developed at Hewlett Packard Laboratories [Youn87].

6.1 Prototype Classes

The HIFI prototype system is developed in the SMALLTALK-80 programming environment [Gold84]. The basic idea, in line with Figure 4.2, is (1) to create a FunctionDescription, (2) to create one or more refinements of the FunctionDescription and (3) to define one or more implementations of the FunctionDescription. The refinement step (2) expresses the original function F, as a composition of functions $f \in f_1, f_2, \ldots, f_n$. The functions f_i may in turn be specified using refinements. Due to this nesting of refinements we can separate the process of implementing a function in two steps: first the generation of a DG, by selecting appropriate refinements for the function F, the functions f_i used in refining it, the functions f_{ii} used in refining the

functions f_i etc. Next the actual partitioning of this DG. The nodes in this SFG may be more or less complex, depending on the depth of the refinements included for generating the DG in the first step. This provides a handle for controlling the granularity of the operations of a node in the implementation.

The major classes to implement a system to design systolic/wavefront architectures in the manner described above will be discussed next.

6.1.1 FunctionDescription:

One of the most important classes of design entities are functions. Functions will be represented by instances of class FunctionDescription. A FunctionDescription bundles together all information about a function. This includes information about the inputs and outputs of the function, the behavior of the function, as well as its refinements. The most important aspect of a FunctionDescription is its capability to create instances. Instances of a FunctionDescription, i.e functions, are used to compute output values given a set of input values. Each output value can be computed independently. To do so the function maintains a memory of the least recently assigned input values. In short, functions are objects that can compute output values and that can set and answer the values of their inputs. In addition functions share a FunctionDescription, that allows access to additional information regarding the instances, e.g. the names and types of the inputs and outputs, the methods used to compute the output values and, from the point of design the most important, a set of refinements and implementations.

To define FunctionDescription's, we use instances of class FunctionOrganizer. A FunctionOrganizer provides access to a set of so-called 'function libraries'. A function library holds a set of function descriptions. Function libraries and FunctionDescription's are assigned names. A particular function can be retrieved by specifying its name and the name of the library it is in. An instance of a FunctionOrganizer can be manipulated by creating a view on it. The functionality for doing this is contained in the class FunctionOrganizerView. A FunctionOrganizerView is shown schematically in Figure 6.1.

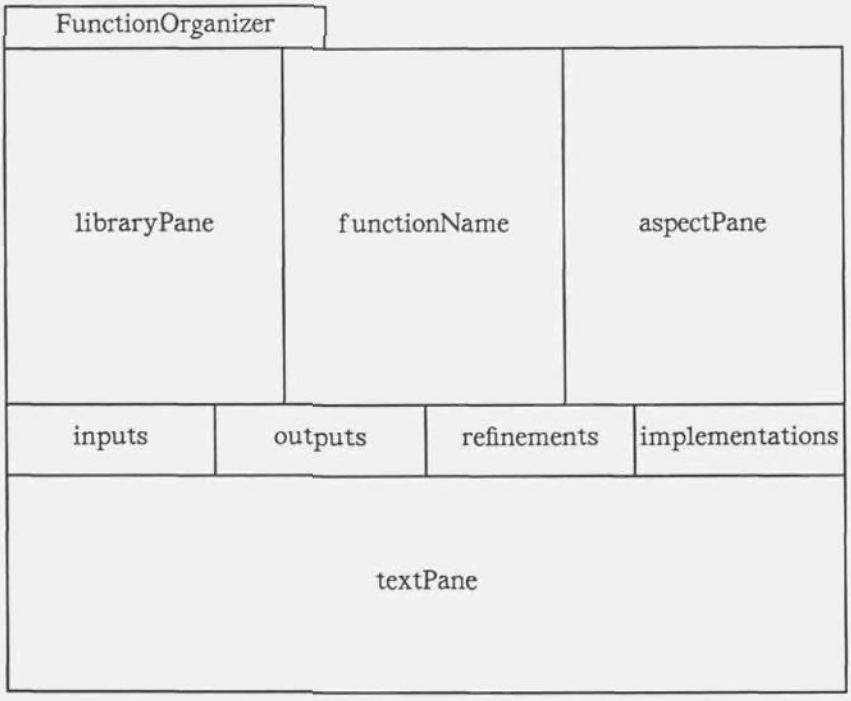

Figure 6.1. Function Organizer View

It consists of a number of subviews or panes, that show various aspects of the FunctionOrganizer. The panes labeled libraryPane and functionName are used to select a FunctionDescription in a particular library. These panes are so-called SelectionInListView's. A name in the list of names displayed in a SelectionInListView is selected by moving the cursor over it and clicking the left mouse button. They communicate with the FunctionOrganizer via a predefined set of messages. The controllers associated with these panes allow the designer to add, remove and rename libraries and FunctionDescriptions respectively. The aspectPane is used to show different aspects of the selected FunctionDescription. The aspect shown can be selected by clicking the left mouse button in one of the panes labeled 'inputs', 'outputs', 'refinements' or 'implementations'. Depending on the selection the aspectPane will show inputs, outputs, implementations, or

refinements. The popup menu associated with the aspectPane also depends on the selection. The designer can always add, remove and rename inputs, outputs, implementations and refinements. In addition, the type of an input or output can be inspected and/or set. When selecting 'show type' a so-called FillInTheBlankView pops up, displaying the type of the selected input or output. The type can be changed by editing the string representing the type. The change is effectuated by 'accepting' it from the popup menu associated with the FillInTheBlankView. A refinement can be entered by selecting 'view' from the popup menu associated with the aspectPane. In order to view a refinement the designer is asked to designate a rectangular area in which a view of the refinement will be displayed. The layout of the view depends on the type of the refinement. In any case it allows the designer to edit all aspects of the refinement. A similar situation exists for the implementations of a function.

For the moment we return to the FunctionOrganizerView shown in Figure 6.1. there is one more pane, i.e. the textPane. The textPane is used for two purposes: (1) in case of inputs, implementations and refinements it is used to display a comment describing the input, implementation, respectively refinement and (2), in case of an output, it displays the text of a method that is used to compute the value of the output. In this method, the names of the inputs can be used as variables. In all cases, a change in the displayed text must be accepted by selecting the item 'accept' from the popup menu associated with the textPane.

6.1.2 FunctionRefinement

Refinements can be added to a FunctionDescription in the aspectPane of a FunctionOrganizerView. When adding a refinement, the designer is asked to specify whether it is a regular refinement. If not, it becomes a structural refinement. Subsequently, the refinement can be inspected by opening a view on it. This is done by selection the command 'view' from the popup menu associated with the aspectPane. The views associated with structural and regular refinements are very different.

6.1.2.1 StructuralRefinement

The StructuralRefinementView allows the DG associated with the FunctionRefinement to be specified directly, i.e. as a composition of functions. The relationship between inputs/outputs of the DG and inputs/outputs of the FunctionDescription being refined are specified as type refinements. The nodes don't need control inputs since they have to perform only one function. This simplifies the definition of the DG considerably. In fact, the view allows the designer to enter a functional expression [Back78] in a graphical way. This is a big advantage, since most of the (apparent) complexity of a functional expression, is due to the fact that a large number of selector functions must be used in order to select the operands for a function from among all other operands. The graphical interface represents the selector functions by the edges connecting inputs and outputs of functions. If necessary, we can generate a functional expression from the graphical representation of the DG.

6.1.2.2 RegularRefinement

The RegularRefinementView allows the DG of the FunctionRefinement to be specified by specifying (1) its index space and (2) a set of dependencies. The index space can be defined in a variety of ways. However, in order to construct the DG, it will be necessary to determine the points belonging to the indexspace, as well as their types, i.e. whether it is an input, output or internal point. In order to specify the dependencies, we have to know the dimension of the index space. A dependency can then be represented as a vector that specifies the difference between the coordinates of the points connected by the dependency. If the dimension of the indexspace is less then or equal to three, the DG can be displayed on a graphics display. This may be helpful when the designer has to select a projection vector later on. In addition to the index space and the dependencies, definition of the refinement requires the definition of an iteration node, a set of input/output (type) refinements and the definition of control values for the control inputs of the DG. The iteration node is specified by defining a set of function bindings. This is most easily done when the designer can open a separate view on the iteration node. This view will allow definition of nodes as defined in section 4.2. The same view can be used when the designer wants

to inspect the a node in a SFG, created from a partitioning of a refinement. The input and output refinements are specified by specifying the relationship between inputs and outputs of the iteration node and inputs and outputs of the FunctionDescription that the refinement describes.

The power and flexibility of regular refinements derives in part from the fact that the index space can be easily parameterized. The implementation of a parameter mechanism is however rather complicated. We will therefore discuss it separately in section 6.1.5. For now we will assume that the points in the index space can be enumerated if necessary.

6.1.3 FunctionImplementation

Implementations can be added to a FunctionDescription in the aspectPane of a FunctionOrganizerView. Subsequently, the implementation can be inspected by opening a view on it. This is done by selection the command 'view' from the popup menu associated with the aspectPane.

The definition of a refinement consists of two steps: (1) construction of a DG by expanding a tree of refinements and (2) partitioning the resulting DG. In order to partition the refinement, the prototype system requires that the DG is regular. In that case, partitioning can be done by choosing a projection vector. Subsequently, the designer may systolize the resulting SFG, in order to create a systolic array that will implement the function described by the FunctionDescription.

6.1.4 Type Definition

In order to define the type of the values carried by the control and data edges, we have to implement a *type definition mechanism*. This mechanism has to support type refinements as well as the definition of the index spaces of regular refinements, parameterized by various properties of a type, e.g. the number of rows of a matrix, or the length of a vector. The types that are important in the context of the prototype system can be classified as either: *scalar*, *tuple*, *vector*, *matrix* or *sequence*. The scalar types are the root of the type hierarchy; they are used to construct tuple, vector, matrix and sequence types. A tuple type is formed by an aggregation of two or more other types, that are the components of the tuple. In order to access the

components, they are named. Vector, matrices and sequence types are formed by a regular arrangement of values of a single other type, the so-called basetype. Vectors and sequences are 1-D collections of data, that have a length. Matrices are 2-D collections of data, organized either in rows, in columns or in diagonals. Several specializations exist, e.g lower triangular, diagonal and upper triangular matrices are distinguished. Matrix, vector and sequence types can be created by specifying their basetype, plus values for the other properties, e.g. the number of rows and columns, the number of diagonals or the length. Tuple types are specified by defining their components, i.e. their name and type. Types can be given a name and added to a dictionary, the so-called TypeCatalog.

Type refinements are viewed as operators that map types in one another. The operators applicable to a type, depend on whether the type is a scalar, a tuple, a vector, a sequence or a matrix. Matrices and vectors can be refined, resulting in sequences of an appropriate type. Sequences can be collapsed, creating vector and matrix types. Matrix, vector and sequence type refinements are mostly used in combination with regular refinements. Tuple types can be decomposed in their components; conversely a collection of types can be aggregated in a tuple.

In order to support the definition of index spaces, whose extent is determined by e.g. the length of a vector, the designer may use the values of the properties of a type. Matrix, vector and sequence types have a standard set of such properties, that can be given values by inheritance, or using the parameter mechanism associated with function descriptions.

6.1.5 Parameter Mechanism

The flexibility and expressiveness of the design system requires the definition of a parameter mechanism. The basic idea is to define a set of parameters for each FunctionDescription. An implementation of a function will ultimately have to assign values to these parameters. An important application of parameters will be in the definition of the index space of a regular refinement.

The functions performed by the nodes in a Dependence Graph can thus have a set of parameters associated with them. The designer will have to specify values for each of them. There are a number of possibilities to do so. First, a parameter may be assigned a numeric value. Second, a parameter may be assigned the value of a parameter of the function being refined, even if that value is not yet known. Third, the parameter may have the value of a property of a type, as discussed in the previous section. The values of these properties can in turn be determined by parameter values, e.g. in previous refinements. The advantage of this mechanism is the high level of expressivity. It is much clearer if a designer can set the value of a parameter to the number of rows in a matrix, then it is to set the value of that parameter to some other parameter, whose meaning may not be clear from the context of the refinement. In addition, the relationship between a parameter value and a type property may be indirect, since the type may have been defined via type refinement.

6.1.6 Verification:

The prototype system allows a design to be verified in a number of ways.

- Type checking: The consistency of a refinement requires that inputs and outputs of nodes connected by edges are of the 'same' type. Similarly, inputs and outputs of functions bound to the same node input or output must be of the 'same' type.
- Correctness: The system will have to be able to evaluate the control functions of the nodes in order to determine the active inputs and outputs of a node. Based on that it is possible to determine the correctness of DG's and SFG's.
- Simulation: Simulation will be an important tool in the verification of the 'functional' correctness of a system, i.e. it is used to determine whether the system indeed computes the desired function. It may also provide insight in various things, such as the required precision, stability etc.

Simulation of data-driven systems is rather easy due to the absence of a global controller. Nodes can be evaluated as long as input data is available

and the output data can be stored. In addition, a global controller may allow the designer to influence the order in which nodes are evaluated [Held87]. The user interface may allow a high degree of interaction between the designer and the simulator, in casu the scheduler.

6.2 HIFI Database

An important aspect that is missing from the prototype implementation discussed in section 6.1 is the database system. A database system will be needed to provide persistent object storage. The objects created and manipulated by the design tools will have to be stored on secondary storage for a number of reasons. Some of these are:

- sharing of data
- backup and recovery
- size

Design systems in general pose a unique set of demands on a database management system, which are not easily obtainable by using existing, commercial database management systems. State of the art commercial DBMS are mostly based on the relational data-model [Date81]. The relational model and the associated relational algebra, provide a firm basis for the design of these systems, but applications of relational DBMS remain limited mostly to those where the data of interest can be easily represented as a collection of tables. One major reason for this is that the abstraction techniques supported by a relational DBMS only resemble the 'aggregation' techniques used by so-called semantic or object-oriented data-models. Aggregation, i.e. the combination of a number of properties or attributes to a new entity, requires a facility for identifying the new entity, so that we no longer have to refer to it in terms of the values of its attributes, but can instead use its id. The DBMS will have to assign a unique id to any entity in the database.

Aggregation is however only one technique used in the semantic and object-oriented data-models that are now being developed. The other important technique is called generalization, or its inverse specialization. Similar to

the class construct in an object-oriented programming language, the definition of types, as made possible by aggregation, allows us to define a type hierarchy as well. A type can be a subtype or specialization of another type, meaning that it inherits all attributes and other properties defined for the type.

The fact that programming languages developed for undertaking large programming projects include the two types of abstraction discussed above, requires the database system to support them as well. The secondary storage provided by an operating system however typically takes the form of an hierarchical (distributed) file system consisting of files and directories. A file can be viewed as an array of bytes that can be extended as necessary. A directory is a special type of file that contains the names of its subordinate files. Directories are used for structuring the file system. This means that in general, there has to be a complicated mapping from secondary storage structure to working memory storage structure.

An object-oriented programming language can provide part of the solution, in that it is possible to equip every class with methods for storing its instances and possibly itself, in a file. Combined with a simple key allocator/deallocator, to make the objects written to the file system unique and retrievable, such a system can be setup without major difficulties. Appendix C describes an object-oriented data-management method setup along these lines. The method described there also takes care of inter-object references. The database is kept consistent in the sense that an object is deleted only, after the last reference to the object is deleted. Reference counting and all other data-management is done transparently to the programmer.

There are two major problems associated with such a simple minded approach. The first is related with efficiency. The method described in Appendix C requires the data-management system to be involved at the same level of detail as the application program. The data-structure present in the database mirrors the working memory data-structure in detail. This implies that if a design tool changes even the smallest part of this data-structure, the database has to be updated. The net result is that the number

of transactions of a design tool with the database, required to keep the database consistent, is enormous. The problem can be alleviated somewhat by allowing extended transactions. Transactions however can't be extended too long because that would reduce multi-user access to the database. Extended transactions also don't increase the granularity of the database objects. In order to really increase the efficiency, it is necessary to wrap a boundary around a collection of volatile objects so that the DBMS can handle these objects as a unit.

The second major problem is associated with the retrieval of objects from the database. The simple system described in Appendix C allows retrieval on the value of the key only. In order to retrieve the data of interest, it is necessary to follow a chain of object references until we reach the desired object. Then, in order to determine whether an object is the desired object, the object has to be fetched, converted to its volatile form and its attributes compared with the attributes of the desired object. A relational database system on the other hand allows the user to send a query, in a high-level query language, to the DBMS. The query is decomposed into simple queries, which are then resolved against the database. The results of the simple queries are combined and send back to the user. The decomposition of a query into a set of simple queries is done so as to reduce the amount of data to be accessed and takes into consideration existing (secondary) indices etc. A relational system can do this and thus take advantage of optimizations, because it has to deal with only one complex type of data, a row of attribute values, specified by a descriptor that provides the types of the attribute values. The types of the attribute values are usually restricted to integer, real, boolean and string.

In the remainder of this section, we will describe an approach that provides a solution for the first problem mentioned above, i.e. that of controlling the granularity of the objects.

6.2.1 Persistent Object Storage and Retrieval

A partial solution providing persistent object storage is provided by the Objective C programming language [Cox86] in the form of a filer mechanism. Any object can be written to and retrieved from a (named) file.

An object is written to a file, by sending its id and the path name of the file to a so-called filer object. The filer object implements the protocol required to convert the object from its volatile form to its persistent form, e.g. the formats used for representing the instance variables. A filer object can restore an object from the file it was written to, if it receives a 'readFrom' message, where the filename serves as the argument. Objects are stored in files, with object id's replaced by record offsets. For example, an ascii representation of a rectangle object represented by two points, respectively its lower left and upper right corners, is given below:

```
0 Rectangle @2 @3
0 Point s 10 s 20
0 Point s 30 s 40
```

Figure 6.2. Symbolic representation of a rectangle

The format used for storing an object is straightforward. Every simple type, e.g. integer, short or double, has its own type identifier character. Pointer to objects (type id) are denoted by the '@' character. If an object contains an instance variable of type id, the object pointed to is recursively included in the file. In this file, every object occupies a record. As a result object pointers can be replaced by record numbers. Given this information, we can easily decode any object. For example, the object represented in Figure 6.2 is a rectangle; the first instance variable is of type id and points to the object stored in record number 2; its second instance variable is also of type id and points to the object stored in the record at position number 3. The point objects each have two instance variables of type short.

In order to generate and interpret these symbolic representations, the filer object needs some information regarding the type of the instance variables of an object. This knowledge is provided by the class descriptions of the objects. Each object contains a pointer to its class object, which contains a so-called classDescriptor string. This string codes the types of the instance variable of the object. Its length corresponds to the number of instance variables.

A filer mechanism as described above is quite powerful, although it has some obvious shortcomings. The most visible is that the file representing an object has to contain all objects that can be reached by following chains of pointers, starting from the object to be stored. This complicates sharing of data between two related though different objects, i.e. objects that one would store in different files. One has to store an entire collection of objects, while it is not possible to preserve relationships between these objects and objects not belonging to the collection. Which objects belong to the collection is determined implicitly by following a chain of pointers. Another disadvantage is that if the objects are small, the overhead of storing them in a file may become substantial.

An easy way out, or so it seems, is to assign every object its own database id and to use this id when filing out the object. When retrieving an object one has to apply the reverse process, i.e. one has to locate the object represented by the id and convert it to volatile form. Such a scheme can be fairly easily combined with a storage manager module similar to the one described in Appendix C. It may also be implemented as an interface to an object oriented database [Anne87].

The problem with this approach is the efficiency. Since all objects get an id, the DBMS has to be involved in retrieving even the simplest objects. A more efficient solution is to assign id's only to those objects that are referenced externally, i.e. that are shared between objects. All other objects can be stored together with the object that references them in a manner similar to what is done when using the filer mechanism. There can be several choices regarding the 'external' objects. The simplest solution is to look at the class of the object. For certain classes we could make all instances accessible via external references. In a large number of applications, such a scheme would suffice. A different scheme would track all references made to an object. If an object would be referenced from within several collections or packages, it would be assigned an external id. For a more detailed discussion, the reader is referred to [Sim87].

7. Examples

In this chapter we will give two more detailed examples of the HIFI design methodology. The first example is the transitive closure algorithm as described by Kung in [Kung86]. The second example defines a system for solving a system of linear equations, using an orthogonal variant of the Faddeev algorithm, developed by [Jain86a].

7.1 Example 1: Transitive Closure

The transitive closure problem can be stated as follows:

Given a directed graph defined by its adjacency matrix a_{ij}, *determine whether there exists a path from a node, say node i, to another node, say node j. If so, the output* $a_{ij}^{+} = 1$, *otherwise* $a_{ij}^{+} = 0$.

The most effective sequential algorithm for the transitive closure problem is the so-called Warshall's algorithm, which can be expressed in single assignment form as follows [Kung86]:

for i,j,k from 1 to N

$$x(i,j,k) \leftarrow x(i,j,k-1) + x(i,k,k-1) \circ x(k,j,k-1)$$

The input is $x(i,j,0) \leftarrow a^{ij}$, the output is $a_{ij}^{+} \leftarrow x(i,j,N)$, + is the logical 'or' operator and o is the logical 'and' operator.

Warshalls algorithm may be rewritten in localized form by adding propagating variables for the row and column variables at each level k as follows:

for i,j,k from 1 to N

$$c(i,j,k) \leftarrow \begin{cases} x(i,j,k-1) & \text{if } j = k \\ c(i,j+1,k) & \text{if } j < k \\ c(i,j-1,k) & \text{if } j > k \end{cases}$$

$$
\begin{aligned}
r(i,j,k) \leftarrow\ & x(i,j,k-1) && \text{if } j = k \\
& r(i+1,j,k) && \text{if } i < k \\
& r(i+1,j,k) && \text{if } i > k \\
x(i,j,k) \leftarrow\ & x(i,j,k-1) + r(i,j,k) \circ c(i,j,k)
\end{aligned}
$$

All dependencies are now constant vectors.

7.1.1 Dependence Graph Design

The above single assignment form naturally leads to a cubic dependence graph. The nodes in this graph compute the connections based on the values of the row and column variables which are propagated depending on the relative magnitudes of the index variables i, j and k, i.e. on the position of the node in the Dependence Graph. Figure 7.1 shows the DG for the case $N = 4$. The edges in the $\pm i$ direction carry the values of the r(i,j,k). Similarly, the edges in the $\pm j$ direction carry the values of the c(i,j,k). The edges in the +k direction carry the values of the x(i,j,k).

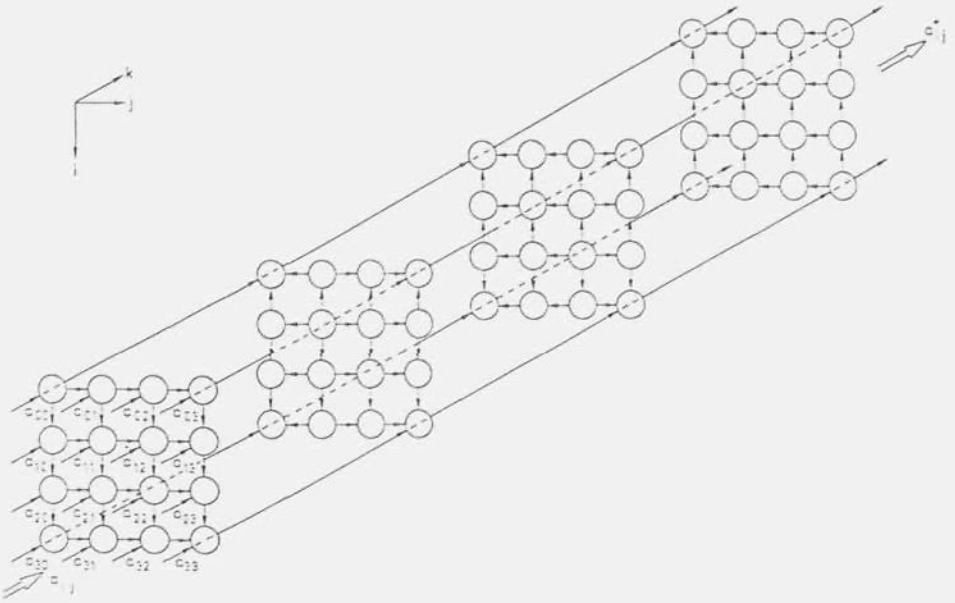

Figure 7.1. Dependency Graph for Transitive Closure

The control required to let the nodes propagate the row and column variables as indicated, can be added by letting control tokens propagate in the k-direction.

7.1.2 Node definition

In this section we will discuss the definition of the nodes used in the DG of Figure 7.1. The basic node is shown in Figure 7.2. All inputs and outputs have been drawn, including the control input and output.

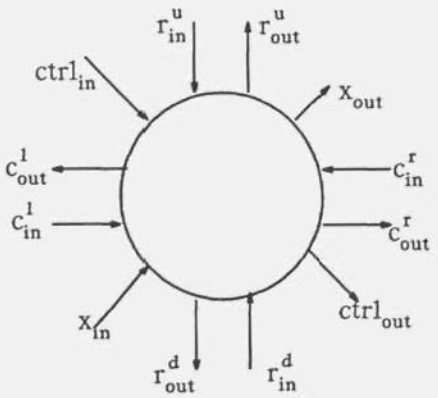

Figure 7.2. Transitive Closure Node

The computation performed by a Transitive Closure (TC) node, is easily defined, by creating a FunctionDescription, say *Transitive Closure*, as discussed in section 6.1.1. Instances of *Transitive Closure* are functions that map a triple of values $<x_{in},c_{in},r_{in}>$ to another triple, $<x_{out},c_{out},r_{out}>$. A FunctionOrganizerView showing the definition of the function *Transitive Closure* is given in Figure 7.3.

The FunctionOrganizerView shown in Figure 7.3, also allows us to define a control function, say *TC control*. In order to define a (DCM) node, we first define a control function for the node. The node can then be created by selecting the command **node** from the menu associated with the function aspect pane (cf. section 6.1.1). When the designer selects this command, a NodeBrowser view pops up, similar to the one shown in Figure 7.4. The NodeBrowser view can be arbitrarily framed and positioned on the display

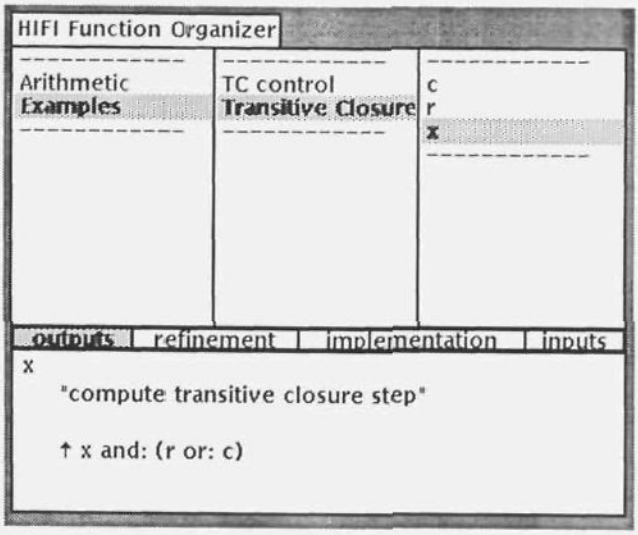

Figure 7.3. FunctionOrganizerView showing the function *Transitive Closure*. The designer has selected the x output; a method for computing the value on the x output is shown in the text-pane in the bottom of the view.

screen, using the window commands that are in a menu that pops-up when clicking the right mouse button.

The designer can select one of four aspects of a node for modification and/or inspection. The selected aspect, e.g. the inputs or outputs, are displayed in the SelectionInList view that occupies the bottum pane of the NodeBrowser view.

An important part of the definition of a node is the definition of a set of so-called FunctionBindings. A FunctionBinding is a mapping from the inputs and outputs of a function to the corresponding inputs and outputs of a node. Every (DCM) node will define a set of such FunctionBindings. The FunctionBindings can be modified and/or inspected by selecting the **binding** aspect in the NodeBrowser. The names associated with the bindings are then displayed in the bottum pane and can be selected. A selected FunctionBinding can be edited, using a so-called FunctionBinding view, as

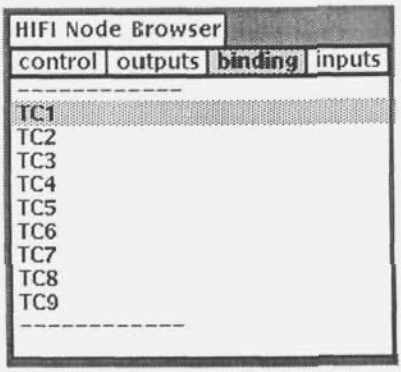

Figure 7.4. NodeBrowser view showing the bindings of the Transitive Closure Node

shown in Figure 7.5.

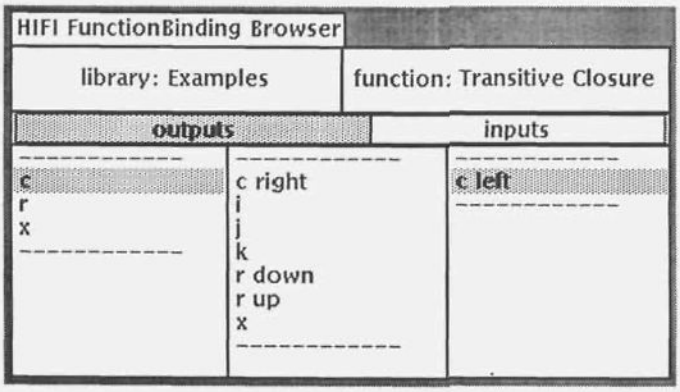

Figure 7.5. FunctionBinding view showing the binding TC1 of the Transitive Closure Node

The control function *TC control* will select one of the FunctionBindings, thus implicitly selecting a function and a set of active (data) inputs and outputs. A function that is to be used as a control function will have to

have an output called **state**. For example, the state output of the function *TC control* can be defined by editing the text-pane of the FunctionOrganizer view shown in Figure 7.6.

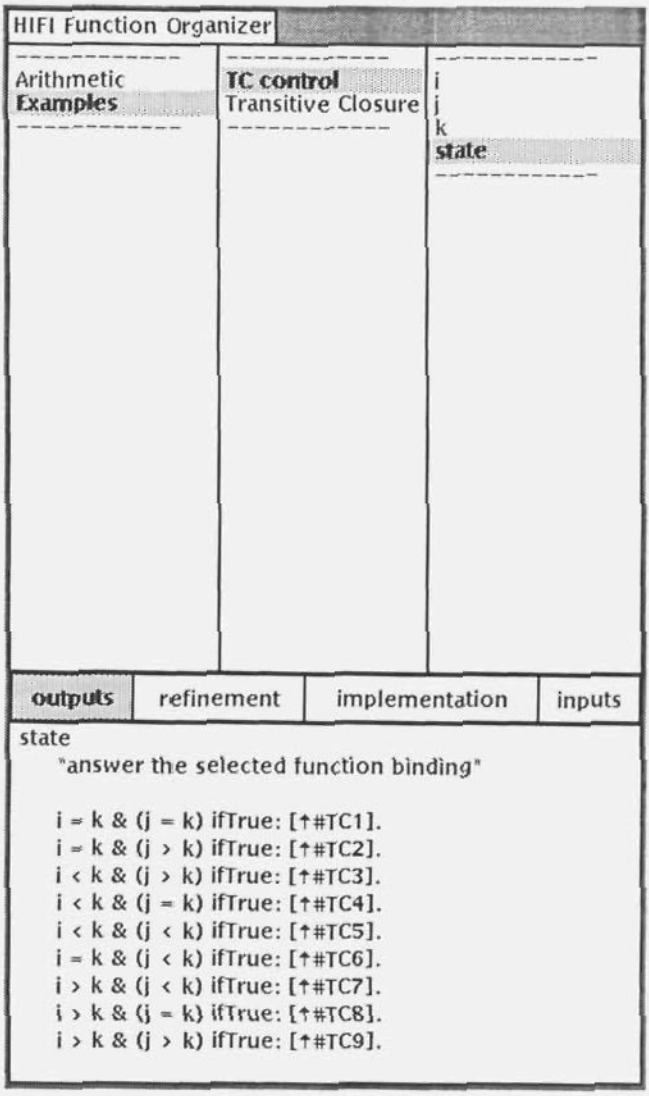

Figure 7.6. FunctionOrganizer view showing the state output of a control function

For the nodes in Figure 7.1 we have to create a FunctionBinding corresponding to every configuration, i.e. selection of active inputs and outputs. In order to limit the number of FunctionBindings we may simplify the DG by assuming that the inputs and outputs of the boundary nodes are connected to source and sink nodes respectively. In that case, nine different FunctionBindings are necessary. Also, the control function can select the appropriate FunctionBinding, by comparing only the values of the i, j and k control indices. Otherwise, the number of FunctionBindings would increase to about 30, while the control function would have to know whether a particular node was on a particular boundary.

In order to see how a FunctionBinding is defined in the prototype, we have to consider the FunctionBinding view already shown in Figure 7.5. The two panes at the top allow the designer to select a function from a library. By clicking the middle mouse button, a menu pops up that allows the designer to select either a library or a function. The functions and libraries that can be selected are those defined by the FunctionOrganizer. Below the two topmost panes there are two panes that allow the designer to select between inputs and outputs. Underneath these aspect selectors, there are three panes, that allow the designer to select inputs and outputs and to bind or unbind them. The leftmost one contains the names of the inputs, respectively outputs of the function selected in the topmost panes. The middle pane contains the names of the input, respectively outputs of the node containing the FunctionBinding. The designer can bind a function input or output to a node input or output by selecting them in the appropriate panes and then selecting the command **bind** from the popup menu associated with these panes. Node inputs and outputs that are bound, are displayed in the rightmost pane, from where they can be unbound.

A schematic description of the FunctionBindings of a *Transitive Closure* node is given below. The names of the node inputs and outputs are shown at the position occupied by the function inputs and outputs in the triples above. Note that more then one node output may be connected to a single function output. The operation evaluating the node can do this; for every evaluation of the function, the eval-operation simply sends the function

output values to the appropriate active outputs. Similarly, it also reads the active inputs only once for every function evaluation.

$$TC_1: \langle x_{in}, x_{in}, x_{in} \rangle \rightarrow \langle x_{out}, c^{r,l}_{out}, r^{u,d}_{out} \rangle$$
$$TC_2: \langle x_{in}, c^{l}_{in}, r^{u}_{in} \rangle \rightarrow \langle x_{out}, c^{r}_{out}, r^{d}_{out} \rangle$$
$$TC_3: \langle x_{in}, c^{l}_{in}, r^{d}_{in} \rangle \rightarrow \langle x_{out}, c^{r}_{out}, r^{u}_{out} \rangle$$
$$TC_4: \langle x_{in}, x_{in}, r^{d}_{in} \rangle \rightarrow \langle x_{out}, c^{r,l}_{out}, r^{u}_{out} \rangle$$
$$TC_5: \langle x_{in}, c^{r}_{in}, r^{d}_{in} \rangle \rightarrow \langle x_{out}, c^{l}_{out}, r^{u}_{out} \rangle$$
$$TC_6: \langle x_{in}, c^{r}_{in}, x_{in} \rangle \rightarrow \langle x_{out}, c^{l}_{out}, r^{u,d}_{out} \rangle$$
$$TC_7: \langle x_{in}, c^{r}_{in}, r^{u}_{in} \rangle \rightarrow \langle x_{out}, c^{l}_{out}, r^{d}_{out} \rangle$$
$$TC_8: \langle x_{in}, x_{in}, r^{u}_{in} \rangle \rightarrow \langle x_{out}, c^{l,r}_{out}, r^{d}_{out} \rangle$$
$$TC_9: \langle x_{in}, c^{l}_{in}, r^{u}_{in} \rangle \rightarrow \langle x_{out}, c^{r}_{out}, r^{d}_{out} \rangle$$

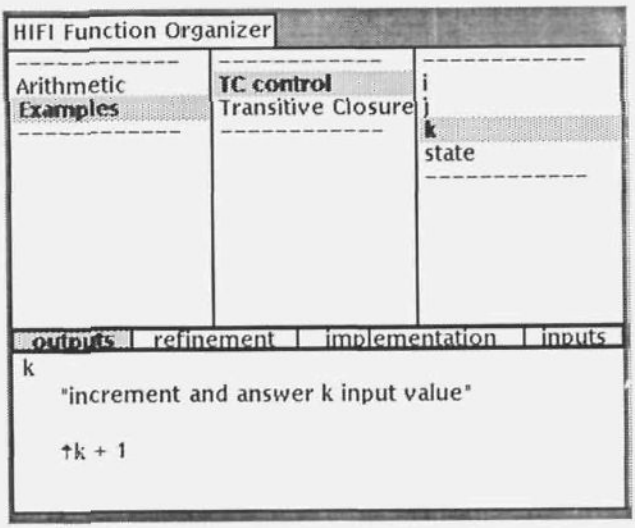

Figure 7.7. FunctionOrganizer view showing the k output of the control function

The control function is defined in the FunctionOrganizer view as shown by Figures 7.6 and 7.7. Figure 7.6 shows the definition of the state output. Similarly, Figure 7.7, shows the definition of the k output, i.e. the output

that increments the control index. Notice that we assume that there are actually three control inputs and outputs, to propagate respectively the i, j and k control values. Alternatively, we could have one control input and output where all three indices would be propagated in a single token.

The DG shown in Figure 7.1 does not offer much choice regarding the projection vector. The only possibility appears to be a projection in the k-direction. The performance of the SFG resulting from such a projection will not be optimal however, due to the fact that the data-dependencies depend on the k index. Other possible projection directions have similar problems. In order to design a more optimal implementation of the transitive closure algorithm we will have to modify the dependence graph itself. Such a modification has been described by Kung [Kung86]. His solution is to reindex the nodes in the DG, so that the dependencies between the nodes in the i-direction become unidirectional. This can be accomplished by moving the first row of each of the DG's shown in Figure 7.1 to the bottom. As a result the dependencies in the i-direction now all point downwards. Moreover, due to the fact that the variables propagated vertically, i.e. the r(i,j,k) are constant, it is not necessary to add global interconnections to the DG. The steps involved in the reindexing are explained in more detail below.

7.1.3 *Reindexing the DG [Kung86]*

If we examine the dependencies in the $k = 2$ plane in Figure 7.1, we find that the variables r(2,j,2) are propagated in both directions in order to update the variables x(i,j,2). If we globally move the first row x(1,j,2) to the (N+1)–st row and relabel it as x(N+1,j,2), then the updating of these variables can be achieved by a uni-directional propagation of the r(2,j,2). However, the variables x(N+1,j,2) also depend on x(1,j,1). In order to localize these dependencies, a new row x(N+1,j,1) should be generated as equal to x(1,j,1) in the previous ($k = 1$) recursion. This can be done if we realize that for given k, the r(i,j,k) variables propagate the value of x(k,j,k−1) in the ± i direction. Since $x(i,j,k)=x(i,j,k-1)+r(i,j,k) \circ c(i,j,k)$, we may conclude that for $i = k$, $x(k,j,k)=r(i,j,k)$. Since we are always shuffling the row for which $i = k$, we may define $x(N+k,j,k)=r(N+k,j,k)$.

Now the x(N+k,j,k) have only local dependencies. The DG after reindexing is shown (schematically) in figure 7.8.

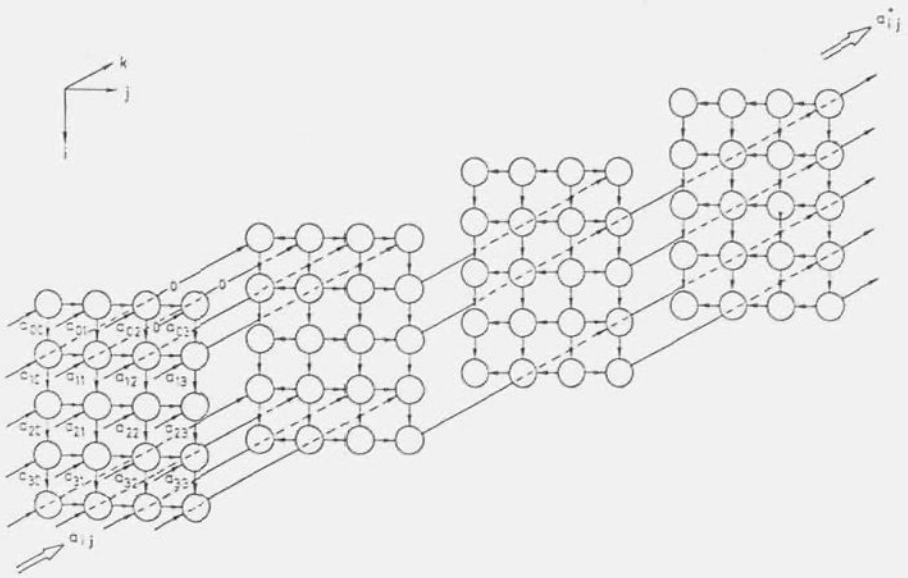

Figure 7.8. DG after reindexing

The number of points in the index space has increased due to the fact that the values of x(N+k,j,k) have to be computed as well. However, since the computation is only a propagation of data values, it can be overlapped (or combined) with the computation of x(k,j,k). This way the throughput rate can be optimal. The control tokens may propagate along the diagonals as well and contain i,j and k indices as before. The number of different states, i.e. function-bindings, of a node is reduced, because the r-values are now propagated in the +i direction only.

7.1.4 Partitioning

By projecting the DG in the +i direction, we obtain a SFG that contains no active cycles. The control sequences are derived automatically. The resulting SFG is shown in figure 7.9. The r(i,j,k) are stored in the buffers associated with the nodes; the i, j and k control values are stored in the buffers between the nodes.

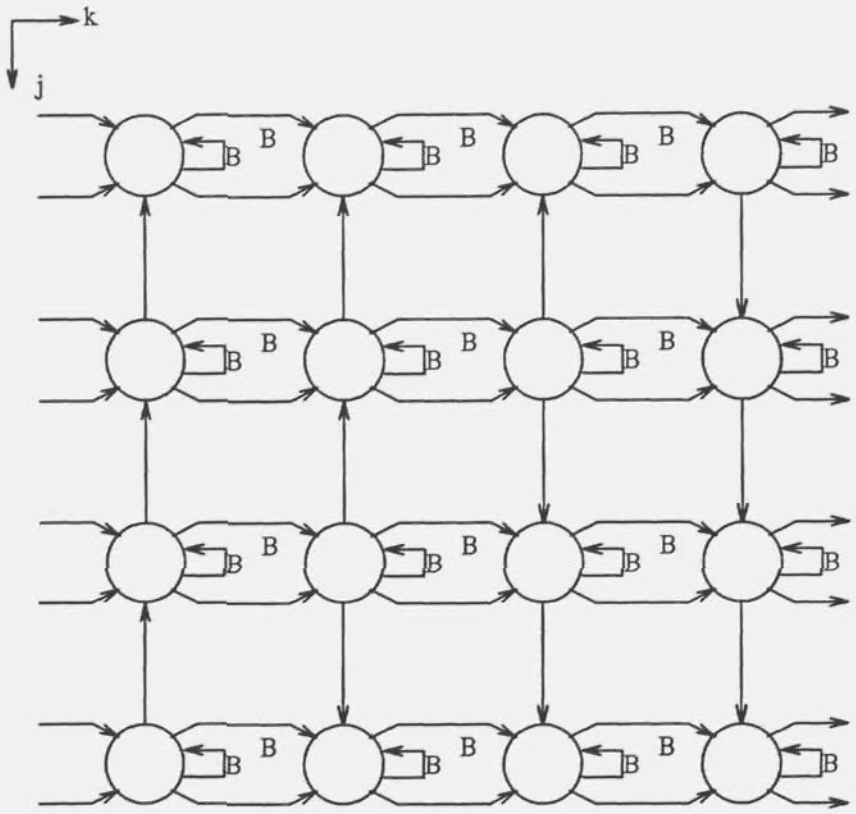

Figure 7.9. Space-Time Partitioning of DG in +i direction

The SFG shown in Figure 7.9 differs from the one given by Kung [Kung86, p. 16], in that the nodes on the diagonal going from lower-left to upper-right are not replaced by pure interconnections. Such a replacement is clearly incorrect, since these nodes have to buffer the value of r(i,j,k) in the corresponding feedback edge, as an inspection of the DG (cf. Figure 7.8) learns.

Further partitioning of the SFG in Figure 7.9 can be achieved by projecting the SFG in the horizontal direction, i.e. with U equal to k. The result is shown in Figure 7.10. In this case no further partitioning is possible, due to the fact that the SFG in Figure 7.10, contains edges in both the + and - j

direction.

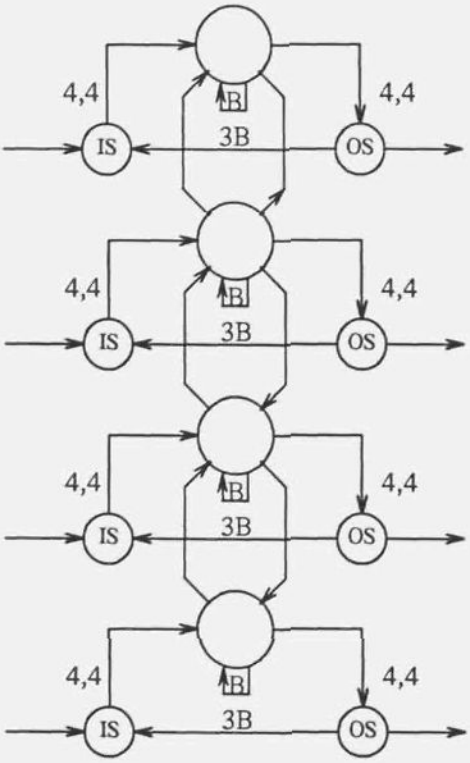

Figure 7.10. STP of SFG in +k direction

7.2 Example 2: Linear Equations Solver

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 data-flow 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 [Jain86a] 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 a VLSI array, after further partitioning.

Thus, given is a system of linear equations A x = b where the matrix A is n×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 = β 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 representing these operations is shown in Figure 7.11. Note however that it does not include the control necessary for the LIFO nodes.

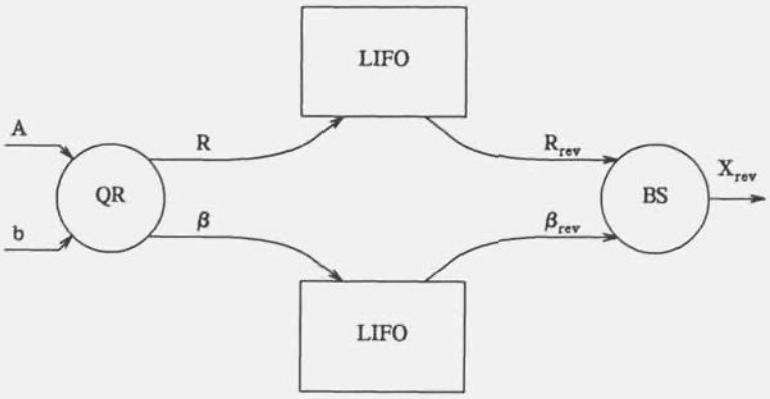

Figure 7.11. Architecture of the Classical Matrix Solver

By a clever arrangement of the data it is, however, possible to restrict the operations to factorization only. Inspired by the work of Faddeev [Fadd59], who presented a Gaussian algorithm which incorporated the backsubstitution, and following [Jain86a], 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 [Wilk71].

7.2.1 The orthogonal Faddeev Algorithm

The more interesting part, from the design point of view, is clearly the factorization procedure, which is easily specified in the form of a LISP procedure *QRFact* (cf. Figure 7.12), which computes and applies a sequence of orthogonal transformations such that the matrix A' is transformed into the upper-triangular form defined above. The procedure returns the last row of the transformed matrix, i.e. the row that contains the solution vector x^t. Each transformation brings one more column of the matrix into the required form. The procedure is defined recursively and is applied to a sequence of successively smaller matrices. The recursion stops if the number of rows of the matrix has become one, in which case the procedure returns the (transformed) elements of the last row. During each recursion, the procedure *QRfact* computes a reflection vector q, that will reflect the elements of the first column of the matrix A', such that all, except the first, become zero. Next it transforms all columns, using the reflection vector q. *QRfact* then strips the transformed matrix of its first row and column, and calls itself recursively on the reduced matrix.

Usage:

In the following procedures 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 *head* and *tail*, where *head* returns the first element of a list and *tail* a list containing all elements, except the first. Another basic LISP procedure is *apply*, which applies a procedure, its first argument, repeatedly to the elements of a set of lists, the remaining arguments, collecting the results in a new list, which is returned as the result of the procedure.

```
(define (QRfact a)
      (let ((q (vector (head a))))
           (let ((a+ (apply (lambda (col) (reflect col q))
                            (tail a))))
              (if (single-row? a)        ; last iteration
                (apply head a+)
                (QRfact (apply tail a+))))))
```

Figure 7.12. Orthogonal Faddeev: The factorization procedure

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 [Abel85].

The factorization procedure uses the procedures *vector* and *reflect*. The procedure *vector* computes Given the vector v, a vector q which represents the elementary reflection matrix.

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

The matrix Q is an operator that transforms v into a vector in the direction of the first unit vector. The procedure *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 in Figures 7.13 and 7.14.

```
(define (reflect x q)
  ; compute the Householder reflection of x with respect to q.
  ; input:  x      - input vector
  ;         q      - head:     norm-square of the reflection vector
  ;                - tail:     reflection vector
  ; output:        - the reflected vector
  (define normq (head q))
  (set q (tail q))
  (define x.q (* 2 (/ (innerproduct x q) normq)))
  (apply (lambda (xi qi)  (- xi (* x.q qi))) x q))
```

Figure 7.13. LISP code for reflect

```
(define (vector a)
  ; compute the Householder reflection parameters
  ; input:
  ;   a      - input vector
  ; output:
  ;         - list head : norm-square of reflection vector
  ;                tail : reflection vector
  (let ((qnorm2 (innerproduct (tail a) (tail a)))
        (a1 (head a)))
       (let ((q (cons (+ a1 (* (if (< 0 a1) 1 -1)
                               (sqrt (+ qnorm2 (* a1 a1)))))
                      (tail a))))
            (cons (+ qnorm2 (* (head q) (head q)))
                  q))))
```

Figure 7.14. LISP code for vector

7.2.2 Design of a system

The first step in the design of a system, which implements the orthogonal Faddeev algorithm, is to define a FunctionDescription (cf. section 6.1.1) representing the algorithm. Starting from this we may define refinements and ultimately one or more implementations.

The algorithm can be represented by a function that maps a matrix A and a vector b, to a vector x, the solution of the equation A x = b (cf. Figure 7.15(a)).

In that case the first refinement (cf. Figure 7.15(b)) will be a structural refinement; it decomposes the function in a part (i.e. a function) for generating the matrix A' and a part representing the factorization of A'. The

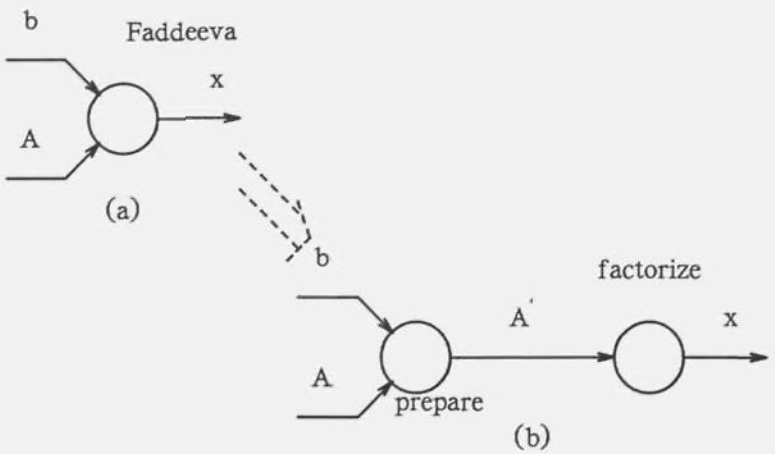

Figure 7.15. Initial design steps

latter function produces a vector representing x. The first refinement may be regarded as an initial step; the function of interest, i.e. the factorization of the matrix, is introduced at this level. The (pre)processing of the input data, represented by the first part of the refinement, as well as subsequent processing of the output data, may be located in a host processor, i.e. it may be thought of as being part of the environment of the system being designed. We will therefore feel free to concentrate exclusively on the factorization.

The next step is to define a regular decomposition of this function. This is suggested by the recursive nature of the function, as shown by the LISP code in Figure 7.12. In this case we define a 2-D regular refinement schematically shown in Figure 7.16.

The nodes used in the refinement are DCM nodes, that can perform both the rotate and the vector function. In addition, they have an inactive state, i.e. a state that selects a function that has no inputs and no outputs and does nothing. The nodes marked T in Figure 7.16 are inactive, i.e. they only propagate the control token; the nodes marked V and R perform a vector, respectively rotate function. The vector function computes the reflection vector q, as discussed above. The rotate function applies the reflection

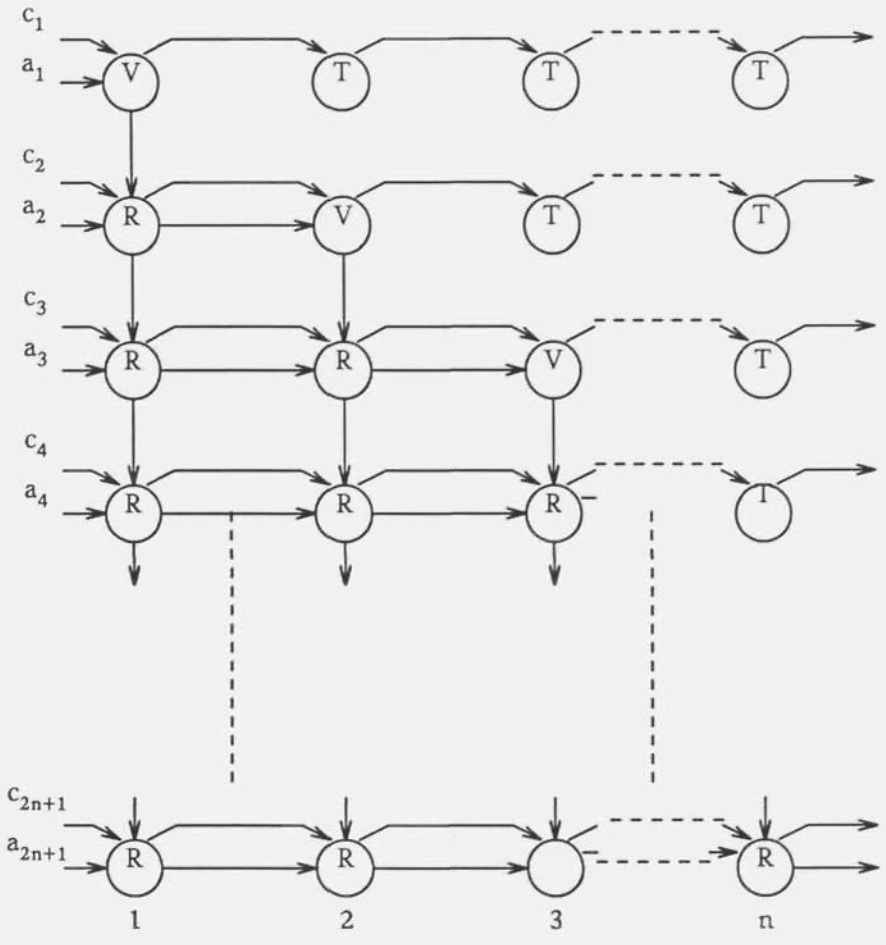

Figure 7.16. 2-D regular refinement of *factorize*

specified by a reflection vector to an input vector. The number of points in the index space associated with the regular refinement, is determined by the dimension n of the A matrix. The index space is defined by 2n + 1 rows and n columns. Note also that the DCM nodes need four states, one more then would be expected, due to the fact that the last row of nodes should not output the reflection vector q.

The control tokens, c_i, are propagated horizontally through the DG. In order to define the control function, we assume that the control tokens are tuples, consisting of a flag and an integer. The flag will be *true*, if the control token propagates along the bottom row. The integer will initially be the number of the row along which the control token is propagating. In order to let each node perform the appropriate function, the control function will decrement the integer. The four states of the nodes then correspond to the following values of the control token:

flag = false, index < 0 : inactive
flag = false, index = 0 : vector
flag = false, index > 0 : rotate_and_propagate q
flag = true, index > 0 : rotate

7.2.2.1 Householder System

In case that a processor capable of implementing both the vector and rotate functions is available, e.g. a so-called Householder processor, the function defined by the DG could be implemented by projecting it on a 1-dimensional SFG. In this case, the projection vector U would be such that it projects the rows of the DG on the (single) row defining the SFG. The resulting SFG is shown in Figure 7.17. The SFG nodes will be similar to the DG nodes in Figure 7.16. Note also that, due to the fact that the control already takes care of the initialization of the buffer, we need not add input and output switches.

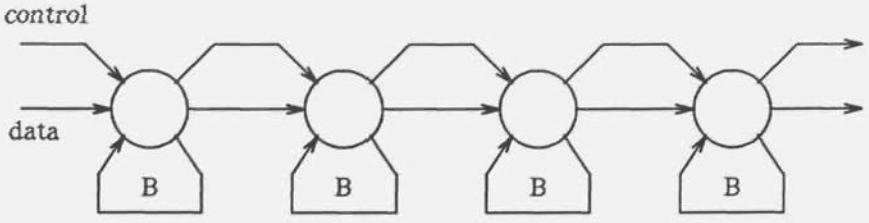

Figure 7.17. Partitioning of *factorize*

The SFG shown in Figure 7.17 can be further partitioned using the method described in section 5.2.3. The resulting SFG, including input and output

switches, is shown in Figure 7.18.

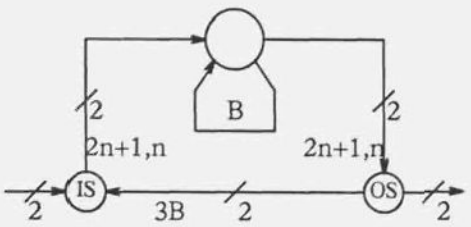

Figure 7.18. Further partitioning of *factorize*

7.2.2.2 CORDIC system

Depending on the level of granularity that the designer wants to capture using a SFG node, it may be necessary to further decompose the vector and rotate functions. Since we want to implement the functions using the same SFG node, it is preferable to use a single DG to specify their decomposition. The differences can then be achieved by changing the control tokens on the control inputs of the DG.

The rotate and vector functions can both be implemented using an array of so-called CORDIC nodes [Vold59]. In a bottom-up fashion, we can define a 1-D array of (n - 1) CORDIC nodes, which can *vector* or *rotate* a vector of n data values, as shown in Figure 7.19.

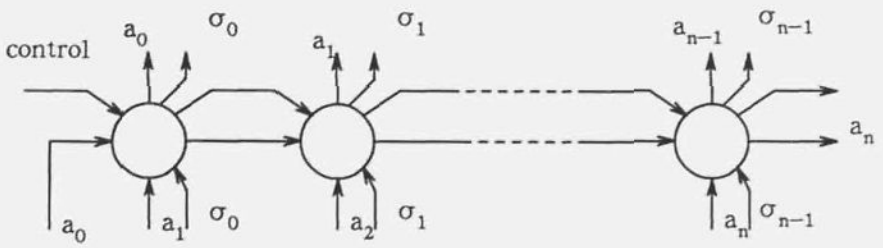

Figure 7.19. Array of CORDIC nodes

The CORDIC nodes are controlled by a single token, that is propagated from left to right (cf. Figure 7.19). The CORDIC nodes themselves perform either a CORDIC-vector or a CORDIC-rotate function on a pair of (scalar)

data values, a_i and a_{i+1}. The σ_i inputs and outputs are used to input/output the rotation parameters. The vector function will compute the σ_i; rotate will input these values use them to rotate the a_i and propagate them unchanged. The function to be performed is determined by the token on the control input. It is possible to specify the implementation of these functions in a regular fashion [Depr84] but that will not be discussed further here.

Substituting an array of CORDIC nodes as shown in Figure 7.19 in the DG shown in Figure 7.16, can be done, if we define the CORDIC array to be a (regular) refinement of both the *vector* and *rotate* functions of the Householder nodes. In this case, that is not an attractive solution, due to the fact that we can simplify the control if we use the CORDIC nodes directly. Therefore, we will define a 3-D regular refinement of the *factorize* function, as shown in Figure 7.20. The arrays of CORDIC-nodes extend in the positive k-direction. The number of nodes in an array depends on the size of the data-vector at its inputs. Because its size is decreasing from left to right, the DG has a relatively complicated shape.
In order to get a regular data-flow, it is necessary to let the CORDIC nodes have an extra input, which is used to enter the first a value (a_0 in Figure 7.19). In order to activate the extra input, instead of the normal input, we have to modify the control, such that the first node selects the extra input. This can be easily achieved by defining an additional control signal, or by extending the number of different control tokens, recognized by the CORDIC nodes. The extra input is used to connect the diagonal dependencies in Figure 7.20.

The shape of the DG and the position of the inputs and outputs complicate the partitioning of the DG, because we are restricted by the fact that the inputs/outputs of the DG have to be projected on inputs/outputs of the SFG. One possible projection vector is however to take U such that the SFG is projected vertically downwards. The resulting SFG is shown in Figure 7.21. Notice that we do not need switches, again due to the fact that the control takes care of a proper initialization of the buffers created by the projection. The nodes shown in Figure 7.21 are similar to those in Figure 7.19, except

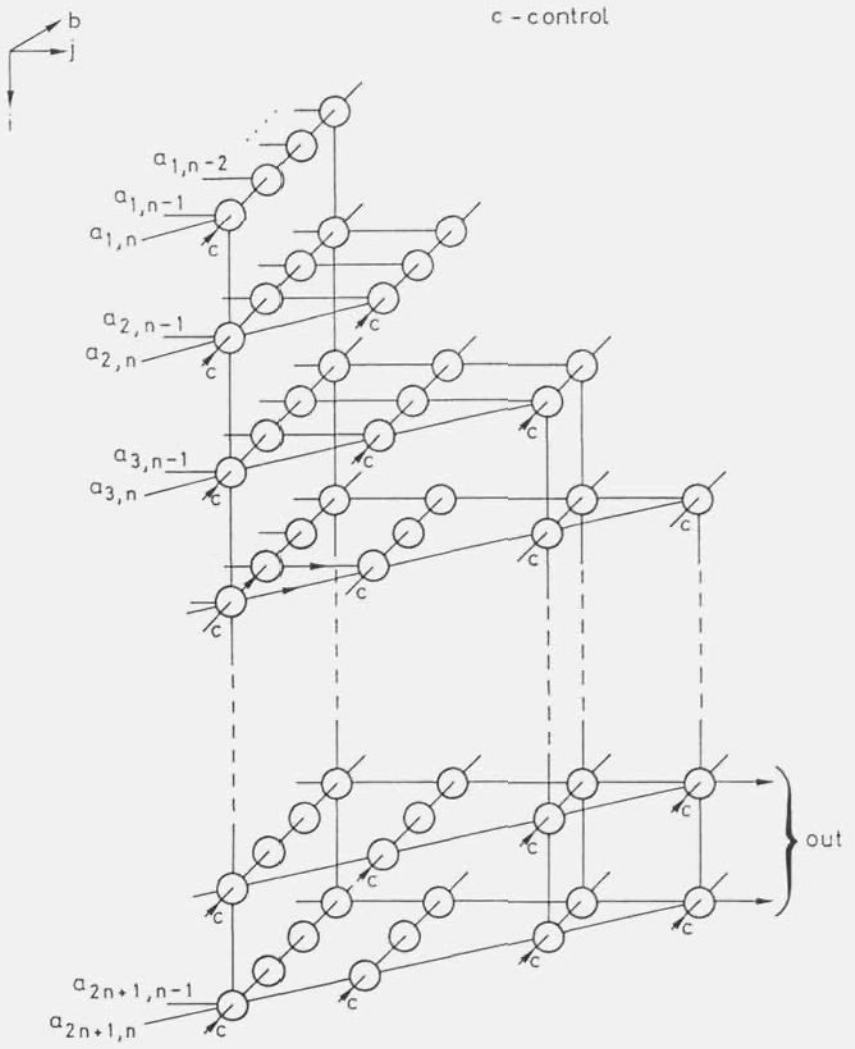

Figure 7.20. 3-D regular refinement of *factorize*

for the fact that they have an extra input and output, as discussed above

and a slightly more complicated control function.

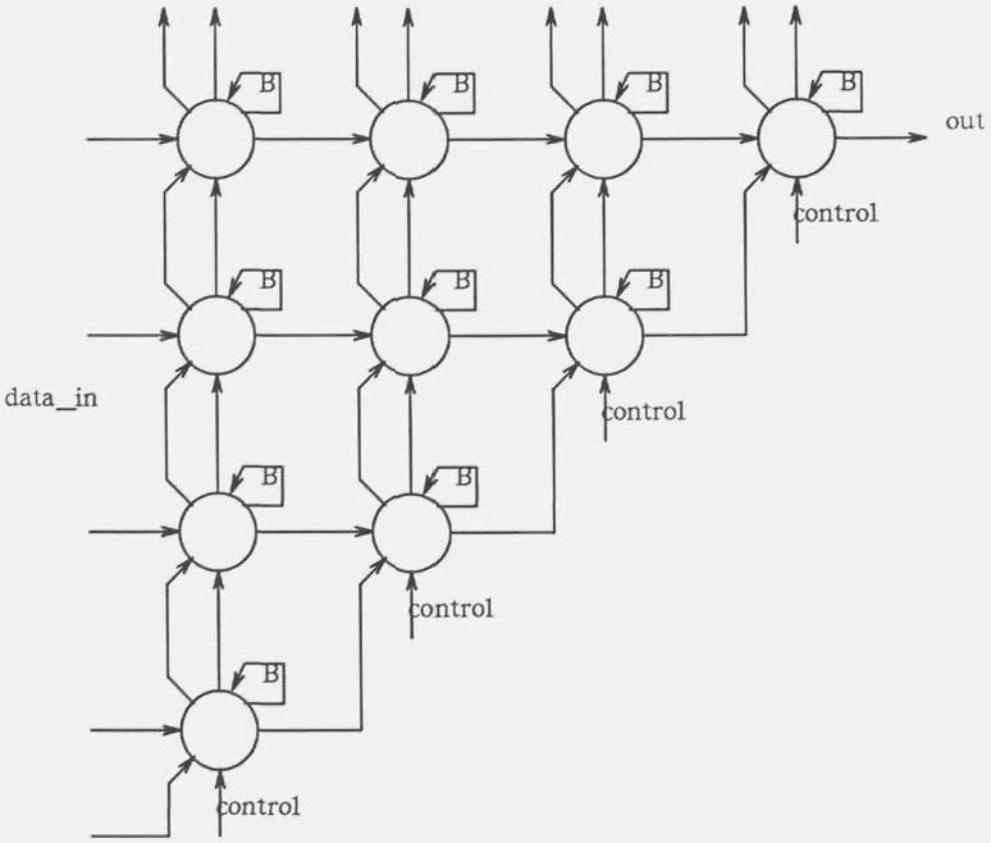

Figure 7.21. Partitioning of 3-D regular refinement of *factorize*

Further partitioning of the SFG shown in Figure 7.21 is possible, but complicated, due to its triangular form and the fact that control and data inputs and outputs are located at all boundaries of the SFG.

8. Discussion

In this thesis we have outlined a new method for designing implementations for a wide class of signal processing algorithms on VLSI processor arrays, such as the wavefront array. The goal of the underlying research was:

- to define a *framework* for systematically transforming and detailing algorithms, until a form is reached that can be implemented on a VLSI processor array without further transformation.
- to develop a set of *tools* that assist in these transformations, that can verify the correctness of the various steps and that provide a 'measure' regarding various design criteria such as efficiency, latency, throughput, hardware requirements etc.
- to *integrate* the tools into a design system.

Due to the large scope of the underlying problems, ranging from algorithmic analysis to issues of software design methodology, database models etc., it will be clear that this thesis is only a first step. From the point of view of satisfying all the above requirements, we are aware that we are still a long way from fulfilling the goals as stated above. We believe however that the design method outlined in this thesis, which is discussed in chapters four and five, is adequate, in the sense that it on the one hand provides a framework for deriving properties of interest of the algorithms being implemented and on the other hand is a suitable basis for the development of design tools. The usefulness of the model is also reflected by the examples discussed in chapter seven. We have shown that, although many details need to be specified, it is possible to specify the implementation of non-trivial algorithms in a systematic fashion. The HIFI design method is however not limited to one particular design style. The generality of the underlying model ensures that different design styles can be accommodated.

The foremost problem that we have encountered is that, in order for the design system to be useful, it must be sufficiently general and flexible to allow definition of all properties that effect the implementation of an

algorithm. The prototype system that we discussed in chapter six allows this, mainly because of the flexibility and the powerful user interface features of the underlying SMALLTALK environment. A strongly related problem is that of design-data management. This was discussed in chapter six as well.

In this chapter we will discuss the HIFI method with respect to the goals as stated above. By identifying the problem areas and by formulating requirements, we will provide some guidelines for the direction of future research.

8.1 Computational Model

The model discussed in chapter four supports the HIFI design method, in that it provides the expressive power and constructs needed in order to define the two HIFI design steps: *refinement* and *partitioning*. The main feature of the model is that it combines process-oriented modeling, with an applicative framework for defining the abstraction and decomposition of a process.

The design process is viewed as consisting of a sequence of *refinement* and *partitioning* steps. The refinement of a function specifies a decomposition of the function in terms of subfunctions. These subfunctions can themselves be refined using a top-down design strategy. The decomposition of a function implies that the data at the function inputs and outputs has to be decomposed as well. This is specified by means of *type-refinements*. A type refinement is a decomposition of a data-type into appropriate subtypes (cf. section 6.1.4). The decomposition of a function is specified in the form of a Dependence Graph (DG), that can be partitioned. Whereas a DG can be viewed as an implementation of a function completely in the 'space' domain, a partitioning of a DG specifies an implementation of a function both in the space and time domain (cf. section 4.1). By the process of partitioning a DG is mapped on a Signal Flow Graph (SFG). Using the HIFI design method, a refinement will be specified by a DG that contains FNC and DCM nodes. These node types were defined in section 4.2.3, as specializations of the more general AST nodes. Since neither FNC nor DCM nodes have an internal

state, we can define a simple procedure for partitioning DG's. To do so, we first partition the nodes of the DG into disjoint classes (*clusters*). Each class can then be mapped on a single SFG node. In case the DG is regular, i.e. if the nodes of the DG correspond to the grid points of an n-dimensional Euclidean subspace, the DG classes are implicitly specified by chosing a projection vector U. All nodes on lines parallel to U belong to the same class and are projected on a single SFG node. However, due to the fact that the functions performed by the DG nodes in a single class may be different, it is necessary that the SFG nodes are able to perform several different functions. There are several different ways to introduce the control required to let the nodes select the appropriate function:

- local control: the nodes have the correct schedule "a priori". This was shown in section 5.2.2, were we partitioned an arbitrary DG. Advantage:

  — no control communication required.

  — simple.

  Disadvantage:

  — control is problem size dependent.

  — further partitioning is not possible.

- distributed control: control is done by having control tokens traveling through the SFG. The most common, used in the examples in chapter 7.

There are many more methods for controlling the functions performed by the nodes in a SFG. One interesting possibility is for example to interleave the control tokens with the normal data-tokens. This requires that the nodes are able to distinguish control and data tokens. The advantage is that a node needs less inputs and outputs. In addition, it may be possible to reduce the number of control tokens, e.g. by assuming that a node performs a particular function on all elements of a sequence of input tokens, until it encounters the next control token. The power of the HIFI model derives in fact from its ability to express a variety of control mechanisms.

If the nodes belonging to the same cluster of DG nodes are interconnected, the partitioning procedure will generate so-called buffers, in order to store the intermediate values. The collection of buffers and the values they contain, form the **state** of the SFG. In order to optimize the SFG, i.e. to reduce the memory requirements of the algorithm, it may be possible to reduce the number of buffers. For example, the SFG shown in Figure 5.8 can do with one buffer less the the three indicated, due to the fact that one of the buffers used to store the output of f_1 can be reused in order to store the output of f_2.

8.1.1 Usefulness of the model for designing Systolic/Wavefront Arrays

The assumption underlying the design method presented here is that any algorithm that can be implemented on a systolic or wavefront array, can be designed using the HIFI system. It can be easily seen that the HIFI method as presented here is indeed capable of designing systolic algorithms. As shown by Rao [Rao85], there corresponds a set of recurrences, to every systolic array. The recurrences can be represented by means of a DG. Since the HIFI method is capable of defining DG's, be it in a slightly different form as that used by Rao, it will be clear that we can indeed design any systolic algorithm. The HIFI system can then be seen as an approach towards embedding the design of systolic algorithms in a framework that allows specification of DG's using the approach of stepwise refinement of functional descriptions.

In the case of a wavefront array, we notice that, by a proper choice of control tokens, the nodes in a SFG can perform any desired function. We can therefore define a SFG, such that the sequence of control tokens on its control inputs, will lead to a sequence of data-graphs, that perform the same computations (function evaluations), as the processors in a wavefront array, during a single recursion. Moreover, such a SFG can be derived by defining a DG, consisting of the data-flow graphs stacked on one another. The dependencies between the different levels of data-flow graphs correspond to the buffers present in the SFG.

We would like to point out here that in our methodology it is not possible for a wavefront array to contain **delay's**. We assume that, prior to the start

of a computation, the array is initialized from the environment. Similarly, at the end of a computation, all data values in the array are send to the environment.

8.2 Design Tools

The prototype system discussed in chapter six outlines the major tools required in a HIFI system. Tools are needed to define:

- function descriptions
- refinements
- partitionings
- types
- type refinements

The approach taken there was inspired by the object-oriented way of developing an application. In order to define objects, we need first model it by an object type, i.e. a *class*. The class definition includes methods for defining and/or modifying all of the objects attributes, and allows creation of instances;, that can be initialized and modified using the methods defined in the class. For example, in order to define a FunctionDescription, we need to define a class FunctionDescription, that allows the attributes of a function , e.g. its inputs, outputs, refinements etc., to be represented and defined. It follows that the tools that are needed to define and/or modify the design objects can be simple. We need a layer that allows the designer to identify objects and send messages to them. The construction of such tools is further simplified if a programming environment such as SMALLTALK is available.

In addition to the tools needed to define design objects, we also need tools to determine properties of design objects or collections of design objects, or to verify their consistency. If the designer has freedom in defining the objects and their attributes, the importance of such tools increases. We can distinguish three important tools here:

- to verify the correctness of DG's.
- to verify the correctness of a partitioning.
- to simulate DG's and SFG's.

We have implemented a simulator that can simulate HIFI SFG's [Held87]. This simulator however, has not yet been integrated into the design environment (cf. section 8.3).

8.3 Design System Integration

A design system becomes much more useful if the tools become part of an integrated design environment. As discussed in chapter one, a design environment consists of three parts, respectively (1) the database and associated database management system, (2) the design tools and (3) the user interface.

In order to achieve a high level of data-independence of the design tools, the data model of the DBMS should match with the object-oriented model used by the design tools. New developments in database models are going in this direction [Fish87]. In section 6.2 we have identified several problems related to the implementation of such database systems.

In order for a design environment to be flexible, it is necessary that it can be extended to provide for the needs of a particular design style. One way of providing this extension capability is by chosing an object-oriented programming language as the implementation language of the design system. Object-oriented languages can be extended easily; one can simply define new methods and/or classes. The prototype system discussed in chapter 6 is based on the SMALLTALK language. In order to customize it for the HIFI system we defined several classes to represent the HIFI design entities.

The biggest advantage of an object-oriented system is that it is possible to build upon a base of existing functionality and applications. When implementing complex software systems this capability can be used to let the capabilities of the design system increase in an evolutionary fashion. Moreover, the tools necessary to extend the system are provided within the

design environment.

References

Abel85. Abelson, H. and Sussman, G., *Structure and Interpretation of Computer Programs*, MIT Press (1985).

Anne87. Annevelink, J., "Objective-C IRIS interface," STL-TM-87-15 Hewlett Packard Laboratories (1987).

Anne88. Annevelink, J. and Dewilde, P., "Object Oriented Data Management based on Abstract Data-types," *Software Practice and Experience (to appear)*, (1988).

Back78. Backus, J., "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).

Beck80. Beckman, F., *Mathematical Foundations of Programming*, Addison Wesley Systems Programming Series (1980).

Bout86. Boute, R., "System Semantics and Formal Circuit Description," *IEEE Trans. on Circuits and Systems* **33**(12) pp. 1219-1231 (December 1986).

Chen83. Chen, M.C., "Space-Time Algorithms: Semantics and Methodology," Ph.D. Thesis, Cal. Inst. of Technology (1983).

Cox86. Cox, Brad J., *Object Oriented Programming: An Evolutionary Approach*, Addison Wesley (1986).

Crem76. Cremers, A. and Hibbard, T., "Formal Modeling of Virtual Machines," *IEEE Trans. on Software Engineering*, (1976).

Crem85. Cremers, A. and Hibbard, T., "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).

Date81. Date, C.J., *An Introduction to Database Systems*, Addison-Wesley Systems Programming Series (1981).

Depr84. Deprettere, E.F., Dewilde, P., and Udo, R., "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).

Dewi86. Dewilde, P. ed., *The Integrated Circuit Design Book*, Delft University Press, Delft, The Netherlands (1986).

Fadd59. Faddeeva, V.N., *Computational Methods in Linear Algebra*, Dover publ. , New York (1959).

Fish87. D. Fishman, "IRIS: An Object-Oriented DBMS," *ACM Trans. on Office Information Systems*, (April 1987).

Gajs83. Gajski, D.D. and Kuhn, R.H., "Guest Editor's Introduction: New VLSI Design Tools," *IEEE Computer* **16**(12)(1983).

Gal-82. Gal-Ezer, R., "The Wavefront Array Processor and its Applications," PhD Thesis, University of Southern California (Dec. 1982).

Gold83. Goldberg, A., Robson, D., and Ingalls, D.H., *Smalltalk-80: The Language and Its Implementation*, Addison Wesley, Massachusetts 01867 (1983).

Gold84. Goldberg, A., *Smalltalk-80: The Interactive Programming Environment*, Addison Wesley, Massachusetts 01867 (1984).

Guer85. Guernic, P. Le, Benveniste, A., Bournai, P., and Gautier, T., "SIGNAL: A Data-Flow Oriented Language for Signal Processing," Publication # 246, IRISA , Rennes, France (January 1985).

Held87. Held, P., "SYSSIM: A Functional Simulator for SFG's," MSc Thesis, Delft University of Technology (1987).

Hilf85. Hilfinger, P.N., "A High-level Language and Silicon Compiler for Digital Signal Processing," *IEEE Custom Integrated Circuit Conference*,

pp. 213-216 (1985).

Hoar85. Hoare, C.A.R., *Communicating Sequential Processes*, Prentice Hall (1985).

Inte86. , "iPSC: Intel Personal Supercomputer," Intel product information (1986).

Inmo86. , "Inmos IMS T424 Transputer," Inmos product information (1986).

Jain86. Jainandunsing, K., "Optimal Partitioning Schemes for Wavefront/Systolic Array Processors," *Proc. IEEE Intl. Conf. on Circuits and Systems*, pp. 940-943 (May 1986).

Jain86a. Jainandunsing, K. and Deprettere, Ed. F., "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).

Jone86. Jones, G. and Luk, W., "Exploring Design by Circuit Transformation ," pp. 91-98 in *Systolic Arrays*, ed. W. Moore et.al., Adam Hilger, Oxford (U.K.) (July 1986).

Kung79. Kung, H.T. and Leiserson, C., "Systolic Array (for VLSI)," *Sparse Matrix Proc.* , pp. 256-282 (1979).

Kung82. Kung, S.Y., Arun, K.S., and Gal-Ezer, R., "Wavefront Array Processor: Language, Architecture and Applications," *IEEE Trans. on Computers* **31**(11)(1982).

Kung83. Kung, S.Y. and Annevelink, J., "VLSI Design for Massively Parallel Array Processors," *Microsystems and Microcomputers* **7** pp. 461-468 (December 1983).

Kung84a. Kung, S.Y., Lo, S.C., and Annevelink, J., "Temporal Localization and Systolization of SFG Computing Networks," *Proc. SPIE*, (August 1984).

Kung84. Kung, S.Y., "On Supercomputing with Systolic/Wavefront Array Processors," *Proceedings IEEE* 72(July 1984).

Kung86. Kung, S.Y., "VLSI Array Processors," in *Systolic Arrays*, ed. W. Moore et.al., Adam Hilger, Oxford (U.K.) (July 1986).

Kung87. Kung, S.Y., *VLSI Array Processors*, Prentice Hall (1987).

Mead80. Mead, C. and Conway, L., *Introduction to VLSI Systems* , Addison Wesley, Reading MA (1980).

Mold86. Moldovan, D. and Fortes, J.A.B., "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," *IEEE Trans. on Computers* C-35(1) pp. 1-12 (Jan. 1986).

Neli86. Nelis, H., Jainandunsing, K., and Deprettere, Ed. F., "Automatic Design and Partitioning of Systolic Arrays," *Tech. Report, Dept. of EE, Delft Univ. of Technology*, (August 1986).

Pete81. Peterson, J.L., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs (1981).

Raba85. Rabaey, J.M., Pope, S.P., and Broderson, R.W., "An Integrated Automated Layout Generation System for Digital Signal Processing," *IEEE Custom Integrated Circuit Conference*, (1985).

Ramm87. Rammig, F.J., *Design Languages: Concepts and Examples*, University of Paderborn (1987).

Rao85. Rao, Sailesh K., "Regular Iterative Algorithms and their Implementations on Processor Arrays," PhD Thesis, Information Systems Lab, Stanford, California (Oct. 1985).

Shee83. Sheeran, Mary, "μFP - An Algebraic VLSI Design Language," Technical Monograph PRG-39, Oxford University (1983).

Sim87. Sim, M., "Object Oriented Data Management," MSc Thesis, Delft University of Technology (1987).

Vold59. Volder, J.E., "The CORDIC trigonometric computing technique," *IRE Trans. Electronic Computers* EC-8 pp. 330-334 (Sep. 1959).

Wilk71. Wilkinson, J.H. and Reinsch, C., *Linear Algebra*, Springer Verlag, New York (1971).

Youn87. Young, C., "Realtime Measurements Graphics (RMG)," Private Communication (1987).

VLSI design for massively parallel signal processors

Recursiveness and locality in signal processing algorithms can be handled with VLSI. **S Y Kung and Jurgen Annevelink*** review the effects of VLSI technology and layout design on processor architectures

In modern signal processing, there are increasing demands for large-volume and high-speed computations. At the same time, VLSI has had a noticeable effect on signal processing by offering almost unlimited computing hardware at low cost. These factors combined have affected markedly the rapid upgrading of current signal processors. We review the influence of the basic VLSI device technology and layout design on VLSI processor architectures. The array processors in which we take special interest are those for the common primitives needed in signal processing algorithms such as convolution, fast Fourier transforms and matrix operations. Regarding VLSI devices, special emphasis is placed on alleviating the burden of global interconnection and global synchronization. For cost-effective design, programmable processor modules are adopted. On the basis of these guidelines, we establish the algorithmic and architectural footing for the evolution of the design of VLSI array processors. We note that the systolic and wavefront arrays elegantly avoid global interconnection by effectively managing local data movements. Moreover, the asynchronous data-driven nature of the wavefront array offers a natural solution to get around the global synchronization problem. The wavefront notion lends itself to a wavefront language (matrix dataflow language (MDFL)) which simplifies the description of parallel algorithms.

microprocessors signal processing parallel algorithms

The ever-increasing demands for high-performance and real-time signal processing necessitate large computation capabilities, in terms of both volume and speed. Therefore the realization of many modern signal processing methods depends critically on high-speed computing hardware. The availability of low-cost high-density fast VLSI devices makes high-speed parallel processing of large volumes of data practical and cost-effective[1]. This presages major technological breakthroughs in realtime signal processing applications. However, the full potential of VLSI can be realized only when its application domains are discriminatingly identified. Traditional computer architecture design considerations are no longer adequate for the design of highly concurrent VLSI computing processors.

For an example of new VLSI design principles, high layout and design costs suggest the use of a repetitive modular structure. Furthermore, the communication has to be restricted to localized interconnections, as communication in VLSI systems is very expensive in terms of area, power and time consumption[1]. A broad and fundamental understanding of the effects of VLSI technology hinges on crossdisciplinary research encompassing the areas of algorithm analysis, parallel computer design, system applications, VLSI layout methodology and device technology.

Our objective is to introduce an integrated research approach to incorporate the vast VLSI computational capability into modern signal processing applications. This should help to shorten the design gap between signal processing theory/algorithm and VLSI processor architecture/implementation. We start, in the next two sections, with a discussion of MOS device technology and VLSI hierarchical design methodology. We then summarize their major influences on processor architecture design. In the same section we discuss the decomposition of (signal processing) algorithms via the concept of a computational wavefront. This leads to the systolic- and wavefront-type array architectures. A wavefront programming language (matrix dataflow language (MDFL)) is introduced and the applications of the systolic and wavefront arrays to signal processing are discussed. Finally we propose future advances in array processor designs.

Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089, USA

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

VLSI DEVICE TECHNOLOGY

Computer technology has been strongly influenced by the technology of device implementation. Until the mid-1960s most signal processing was performed with specialized analogue processors because of the complex hardware, the power consumption and the lower speed of conventional digital systems. Today's VLSI offers a much greater density of components, higher speed and lower power use than those of most leading analogue devices. In order to understand fully the effect of VLSI on computer technology, it is necessary to examine the characteristics of the device technology.

MOS building blocks

MOS circuits are typically made of n-channel enhancement and depletion transistors (NMOS) or n-channel and p-channel enhancement transistors (CMOS). The basic MOS building block is the inverter circuit. Figure 1 shows the basic NMOS and CMOS static inverter structure. Usually the transistor connected to ground is referred to as the pull-down transistor, while the transistor connected to V_{dd} is referred to as the pullup transistor. The pullup transistor in the NMOS inverter is a depletion mode transistor, ie it will always be on. The pullup transistor of the CMOS inverter

 0141–9331/83/100461–08 $03.00

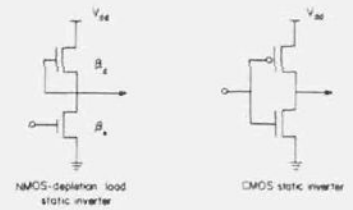

Figure 1. Basic MOS inverter structure

(cf. Figure 1) is a p-channel enhancement type; it will be on only when the voltage at the gate is low with respect to the voltages at the drain and source.

For the design of a static NMOS inverter the aspect ratio, defined as the quotient of β_E and β_D, is an important parameter. (Since $\beta = (W/L)\beta_0$ the aspect ratio is simply the quotient of the W/L ratios of the enhancement and depletion transistor.) To ensure that the low output voltage of the inverter is below the threshold voltage of the enhancement transistor, the aspect ratio must be around 3. In this case the output of the inverter can be used to drive the input of the next inverter.

The output voltage of the CMOS inverter can be changed from V_{dd} to V_{ss} independently of the size of the p-channel and n-channel transistor. However, the area necessary to lay out a CMOS circuit is larger than that for a corresponding NMOS inverter, because of the space needed for the wells to make both n-channel and p-channel transistors in the same substrate. (A well is an n-type or p-type diffusion region, used to make p-channel or n-channel transistors in a p-type or n-type substrate respectively.)

From the basic inverter circuit we can derive the more complex logic functions. For example, by replacing the pull-down transistor of the NMOS inverter by a serial and parallel combination of enhancement transistors we can form any desired logic function.

Power dissipation

A major difference between NMOS and CMOS is the power dissipation. A CMOS inverter draws power only in a transient condition, because the gates for the p- and n-channel transistors (cf. Figure 1) are directly connected. This implies that normally only one of the transistors will be on. Thus the major power consumption is proportional to the switching frequency. An NMOS inverter, in contrast, draws power whenever the pulldown transistor is conducting. Since a typical NMOS circuit will on average have 50% of its gates on, it will draw orders of magnitude more power than an otherwise comparable CMOS circuit.

Scaling

Scaling of the technology, ie reducing the minimum feature size, is the driving force behind the increase in the device density of integrated circuits in the past 10 years. However, with increasing device densities the side effects of scaling become increasingly more critical. One such effect is the increased importance of the interconnection delay: in the submicron technology, the interconnection delay will become comparable with or even greater than the switching delay of the logic gates. This is because the line response time RC is unchanged by scaling while the switching speed of the logic gates is scaled down by a factor k when the power supply is also decreased. (The switching speed is decreased by a factor k^2 while the power supply voltage remains constant.) Moreover, if the average interconnection length is not scaled down by the same factor k, the interconnection delay may even increase, further aggravating the situation. (It should be noted here that for typical VLSI circuits the average interconnection length does not scale down by a factor k because a large number of interconnections still have to cross the entire chip.)

VLSI LAYOUT DESIGN METHODOLOGY AND CAD TOOLS

VLSI design differs from existing hardware and software design in that layout considerations are more important since the layout affects the interconnection lengths and hence both the cost and performance (cf. the last section) of the resulting circuit. Hence floorplanning is an important aspect of VLSI design. Another major concern in VLSI design is the complexity associated with designing a circuit consisting of 50 000 devices (as in today's technology), or even up to 1M devices in the future[2]. To cope with this complexity a hierarchical design methodology is of vital importance. We first review the concepts of floorplanning and different description levels, which constitute the main body of the hierarchical design methodology. We then discuss the CAD tools, especially simulation and verification programs, developed to assist designers.

Hierarchical design methodology

Description levels

To keep the design description comprehensible it is necessary to introduce different levels of system description. Usually we consider four description levels:

- architectural level
- register/logic level
- electrical/circuit level
- geometric/layout level

The major complication with this approach in VLSI design is that all the levels will be interwoven. This is especially clear in leaf-cell design, which combines electric transistor circuit design and layout design. These two levels are difficult to separate because the parasitics that affect the electrical behaviour depend strongly on the layout design.

Floorplanning

The importance of floorplanning stems from the importance of global topology optimization, rather than local optimizations, for reaching a high device density and avoiding long interconnections. In general it is necessary to draw up a floorplan at the earliest stage, ie based on the architectural specification[3]. This floorplan should be based on the best prediction of the sizes and shapes of the main functional blocks and, at the same time, take into account their interconnections and consider the distribution of the power and clock lines. This is often a very demanding task, requiring a broad knowledge of VLSI design.

In general the floorplanning is also done hierarchically. For instance the initial floorplan contains blocks representing functions such as datapath, control and memory. Subsequently the floorplans of these functional blocks are drawn. They in turn consist of blocks, eg the datapath consists of a register block, an arithmetic logic unit block etc. The important thing here is that the more detailed

floorplans must adhere to the topological constraints imposed by the higher-level floorplans.

The complexity of the VLSI design depends very much on how a design can be decomposed into lower-level blocks and on the number of different blocks. A good design relies on a clear and regular floorplan and well defined interfaces between the functional blocks to achieve both a short design time and a high device density. In a pure top-down design strategy, we use successively lower-level description languages to describe functional blocks in a successively more refined floorplan. In practice this may not be effective; for instance, in an early phase of the design we may need to know whether a certain topology is in fact realizable. In that case we may need to have the more detailed (lower-level) knowledge for that part of the design. In other words, to arrive at an optimal design it is inevitable that we use an iterational top-down/bottom-up design strategy.

CAD tools

Relative to today's CAD needs, CAD tools for VLSI design are very much in a rudimentary stage of development. Most of these tools still require extensive interaction between the CAD software developers and the VLSI hardware designers. However, the actual process of designing the layout of a VLSI chip is fairly well supported at the moment. Interactive layout editors and design rule checkers relieve the designer from most of the tedious work of specifying and checking the layout.

Another area in which CAD programs are relatively frequently used is that of simulation and verification. Below we briefly describe some of these programs.

Architectural simulation

One of the major tools in use for architectural simulation is the instruction set processor specification (ISPS) simulator. This tool is based on the ISPS description language. ISPS describes the external structure and the behaviour of hardware units. The behavioural aspects are described by procedures, so that many of the structured programming techniques developed for high-level programming languages can be put to use in digital hardware design. An overview of the ISPS notation and its applications has been given by Barbacci[4].

Register transfer/logic simulation

A register transfer language is used to describe the actual hardware in terms of registers, as well as dataflows and operations on data between registers. A very useful type of simulator results when the register transfer simulation is combined with more detailed gate-level simulation. An example of such a 'mixed-mode' simulator is RTsim[5]. This simulator combines register transfer simulation with switch-level simulation by allowing the replacement of a functional unit by a transistor network. The switch-level concept, developed by Bryant[6], allows the modelling of the bidirectional pass transistor. Switch-level simulators are often used in MOS circuit design since they combine efficiency with detailed gate-level simulation.

Circuit simulation

The circuit simulation program Spice is widely used, in both industries and universities, for the simulation of electrical circuits. Spice computes the voltages and currents in an electrical circuit with a precision better than 0.1%. However, this high precision (due to the precise modelling of the circuit elements) means that we cannot simulate more than between 200 and 500 transistors at once.

DESIGN OF VLSI ARRAY PROCESSORS: ALGORITHM, ARCHITECTURE AND LANGUAGE

Although there have been major advances in VLSI device technology, the main thrust of VLSI depends critically on novel architectural designs. VLSI technology makes possible the realization of supercomputer systems as well as special-purpose array processors, capable of processing thousands of operations or floating point operations per second. In order to achieve such increases in throughput rate, the only effective solution appears to be massively parallel array processors.

Influence of device technology and layout design on VLSI architecture

An array processor is composed of an array of processor elements which are interconnected directly or indirectly. The design of special-purpose array processors should be based on the potential of the VLSI device technology and design methodology and the constraints imposed by them. The key aspects to consider are

- interconnection
- system clocking
- modularity
- programmability

Interconnections

Optimization of the interconnection patterns (between the processor elements) is a key to the full realization of VLSI computing power. There are several important aspects of VLSI interconnections.

Local communications. Interconnections in VLSI systems are implemented in a two-dimensional circuit layout with few crossover layers. Therefore communication has to be restricted to localized interconnections[2,3].

Scaling effect. When the technology is scaled down we must be able to scale down the interconnections by the same factor. Otherwise the *RC* line delay will limit the performance of the system (*cf.* the section on VLSI device technology).

Static versus reconfigurable interconnections. For signal processing array processors, a static mapping of activities onto processors may prove more efficient and cost-effective without creating an undue loss of flexibility. However, reconfigurable interconnections can be used to improve the fault tolerance of the system.

System clocking

The timing framework is a very critical issue in designing the system, especially for large-scale computational tasks. Two opposite timing schemes come to mind, namely the synchronous and the asynchronous timing approaches. In the synchronous scheme, there is a global clock network which distributes the clocking signals over the entire chip. The asynchronous scheme can be implemented by means of a simple handshaking protocol. For large-scale systems the asynchronous scheme is clearly desirable[7].

Modularity

Large design and layout costs suggest the use of a repetitive modular structure. To implement the modules we want to make best use of the device technology. Thus we have to identify the primitives that can be implemented efficiently. The complexity of the primitives depends on a number of factors, eg prior design experience (the basic-cell library) and available CAD tools (eg a programmable logic array generator). In designing the primitives we try to find simple and regular structures with a flexible topology that are easy to fit in the design. It is often less expensive in terms of area, delay and power dissipation to implement a general function than to implement a specific function, while if a general function can be implemented the details of its operation can be left unbound till later, providing a much cleaner design interface.

Programmability

In addition, it is very important to make a processor programmable so that the high design cost may be amortized over a broader market basis. However, there is a tradeoff between the flexibility of a processor and its hardware complexity. From a top-down point of view, we should exploit the fact that a great majority of signal processing algorithms possess a recursiveness and locality property (*cf.* the next subsection). Indeed, a major part of the computational needs for signal processing and applied mathematical problems can be reduced to a basic set of matrix operations and other related algorithms[8]. This commonality can be identified and then exploited to simplify the hardware and yet retain most of the desired flexibility.

Parallel algorithm analysis

To exploit effectively the potential concurrency present in many modern signal processing algorithms, a new algorithmic design methodology is needed. Concurrency is often achieved by decomposing a problem into independent subproblems or into pipelined subtasks and varies significantly among different techniques. An effective algorithm design should start with a full understanding of the problem specification, signal mathematical analysis, (parallel and optimal) algorithmic analysis, and then mapping of algorithms into suitable architectures. The effectiveness of (static) mapping of activities onto a processor array is directly related to the decomposability of an algorithm. Moreover, the preference for regularity and locality will have a major influence in deriving parallel and pipelined algorithms. In our work, the two most critical issues – parallel computing algorithms and VLSI architectural constraints – are considered

- to structure the algorithm to achieve the maximum concurrency and therefore the maximum throughput rate
- to cope with the communication constraint so as to have the best range of processing throughput rates

To conform with the constraints imposed by VLSI, we now look into a special class of algorithms, ie recursive and locally (data) dependent algorithms. (In a recursive algorithm, all processors do nearly identical tasks and each processor repeats a fixed set of tasks on sequentially available data.) A recursive algorithm is said to be local if the space index separations incurred in two successive recursions are within a given limit. Otherwise, if the recursion involves globally separated indices, the algorithm is said to be global and it will always call for globally interconnected computing structures.

However, the assumption of recursive and locally data-dependent algorithms incurs little loss of generality, as a great majority of signal processing algorithms possess these properties. One typical example is a class of matrix algorithms, which are most useful for signal processing and applied mathematical problems. The concept of wavefront computing originates from algorithmic analysis; it will lead to a coordinated language and architecture design. In fact, the algorithmic analysis of, say, the matrix multiplication operations will lead first to a notion of two-dimensional computational wavefronts.

We shall illustrate the algorithmic analysis, leading to the wavefront concept and a coordinated architecture and language design by means of a matrix multiplication example. Let

$$A = [a_{ij}]$$

$$B = [b_{ij}]$$

and

$$C = A \times B$$

all be $N \times N$ matrices. The matrix A can be decomposed into columns A_i and the matrix B into rows B_i. Therefore

$$C = [A_1 B_1 + A_2 B_2 + \ldots + A_N B_N] \quad (1)$$

The matrix multiplication can then be carried out in N recursions

$$c_{ij}^{(k)} = c_{ij}^{(k-1)} + a_i^{(k)} b_j^{(k)} \quad (2a)$$

$$a_i^{(k)} = a_{ik} \quad (2b)$$

$$b_j^{(k)} = b_{kj} \quad (2c)$$

for $k = 1, 2, \ldots, N$, and N sets of wavefronts are involved.

Pipelining of computational wavefronts

The computational wavefront for the first recursion in matrix multiplication is now examined.

A general configuration of computational wavefronts travelling down a processor array is shown in Figure 2. The wavefronts are similar to electromagnetic wavefronts. Each processor acts as a secondary source and is responsible for the propagation of the wavefront. Pipelining of the wave-

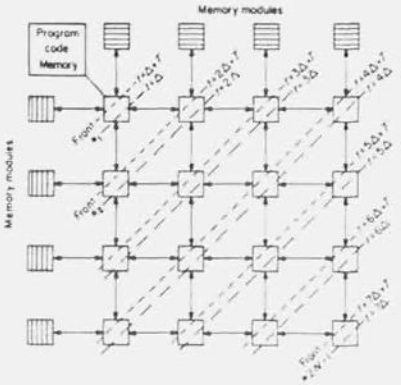

Figure 2. Two-dimensional wavefront array (——, first wave; ····, second wave; Δ, unit time of data transfer; T_i unit time of arithmetic operation)

fronts is feasible because the wavefronts of two successive recursions will never intersect (Huygens' wavefront principle), as the processors executing the recursions at any given instant will be different, thus avoiding any contention problems. We note that the correctness of the sequencing of the tasks in the individual processor elements is essential for the wavefront principle.

Suppose that the registers of all the processing elements are initially set to zero, ie

$$c_{ij}^{(0)} = 0 \text{ for all } i, j$$

The entries of A are stored in the memory modules to the left (in columns), and those of B in the memory modules on the top (in rows). The process starts with processor element (1, 1)

$$c_{11}^{(1)} = c_{11}^{(0)} + a_{11}\, b_{11}$$

is computed. The computational activity then propagates to the neighbouring processor elements (1, 2) and (2, 1), which execute in parallel

$$c_{12}^{(1)} = c_{12}^{(0)} + a_{11}\, b_{12}$$

and

$$c_{21}^{(1)} = c_{21}^{(0)} + a_{21}\, b_{11}$$

The next front of activity will be at processor elements (3, 1), (2, 2) and (1, 3), thus creating a computational wavefront travelling down the processor array. It should be noted that wave propagation implies localized dataflow. Once the wavefront sweeps through all the cells, the first recursion is over (*cf.* Figure 2).

As the first wave propagates, we can execute an identical second recursion in parallel by pipelining a second wavefront immediately after the first one. For example, the (i,j) processor will execute

$$c_{ij}^{(2)} = c_{ij}^{(1)} + a_{i2}\, b_{2j}$$

and so on. It is possible to have several different kinds of wavefront propagation. The only critical factor is that the order of task sequencing must be correctly followed.

Systolic array

Systolic processors[9,10] are a new class of digital architectures that offers a new dimension of parallelism. The principle of systolic structure is an extension of pipelining into more than one dimension. According to Kung and Leiserson[9], 'a systolic system is a network of processors which rhythmically compute and pass data through the system.' For example, they showed that some basic 'inner product' processor elements ($Y \leftarrow Y + A * B$) can be locally connected to perform finite impulse response filtering similarly to the transversal filter. Furthermore, two-dimensional systolic arrays (of the inner product processor elements) can be constructed to execute efficiently matrix multiplication, logical unit decomposition and other matrix operations.

The basic principle of systolic design is that all the data, while being 'pumped' regularly and rhythmically across the array, can be effectively used in all the processor elements. The systolic array features the important properties of modularity, regularity, local interconnection, and highly pipelined highly synchronized multiprocessing.

A detailed description of data movements and computations in a systolic array is often furnished in terms of 'snapshots' of the activities. For the matrix multiplication example, the input data (from matrices A and B) is prearranged in an orderly sequence. The output data (of the matrix C) is pumped from the other side of the array, meeting the right data and collecting all the desired 'products'. For more details, the reader is referred to the articles by Kung and Leiserson[9,10].

However, there are several unresolved controversial issues regarding systolic arrays. First, (pure) systolic arrays tend to equalize the time units for different operations. As an example, for the convolution systolic arrays in Kung and Leiserson's work[9] a local data transfer causes the same time delay as a multiply and add, ie one full time unit. This often results in unnecessary waste of processing time, since the data transfer time needed is almost negligible. This motivates what we have called multirate systolic arrays[11]. More critically, the systolic array requires global synchronization, ie global clock distribution. This may cause clock-skew problems in implementations of high-order VLSI systems. Another issue of concern is ease of programmability for complex dataflows in systolic-type arrays. These problems gave rise to the notion of a wavefront array[12], based on asynchronous data-driven schemes in dataflow machines.

Wavefront array

A wavefront array is a programmable array processor[12]. It differs from a systolic array in that the data communication between adjacent processors is asynchronous. Thus it combines the asynchronous data-driven properties of dataflow machines with the regularity, modularity and local communication properties of systolic arrays. There are at least two distinct advantages associated with the wavefront array.

- The wavefront architecture circumvents the need for global synchronization.
- The wavefront language MDFL offers an effective space–time programming language.

Array size: $N \times N$

Computation: $C = A \times B$

k^{th} wavefront: $c_{ij}^{(k)} = c_{ij}^{(k-1)} + a_{ik}\, b_{kj}$, $k = 1 \ldots N$

Initial: Matrix A is stored (row by row) in the memory module on the left
Matrix B is stored (column by column) in the memory module on the top

Final: The result is in the C registers

```
begin
  set.count N;
  repeat
    while wavefront in array do
    begin
      fetch B, UP;
      fetch A, LEFT;
      C: = C + A * B;
      flow A, RIGHT;
      flow B, DOWN;
      move R, D;
    end
    decrement.count;
  until terminated;
end
```

Figure 3. Example of MDFL program for matrix multiplication

Wavefront language

The wavefront notion helps greatly to reduce the complexity in the description of parallel algorithms. The mechanism provided for this description is the special-purpose wavefront-oriented language termed MDFL[12]. The wavefront language is tailored towards the description of computational wavefronts and the corresponding dataflow in a large class of algorithms (which exhibit the recursivity and locality mentioned earlier). Rather than requiring a program for each processor in the array, MDFL allows the programmer to address an entire front of processors. In contrast with the heavy burden of scheduling, resource sharing and control of processor interactions that is often encountered in programming a general-purpose multiprocessor, the wavefront notion can facilitate the description of parallel and pipelined algorithms and drastically reduce the complexity of parallel programming. To translate the global MDFL into instructions for the process elements,

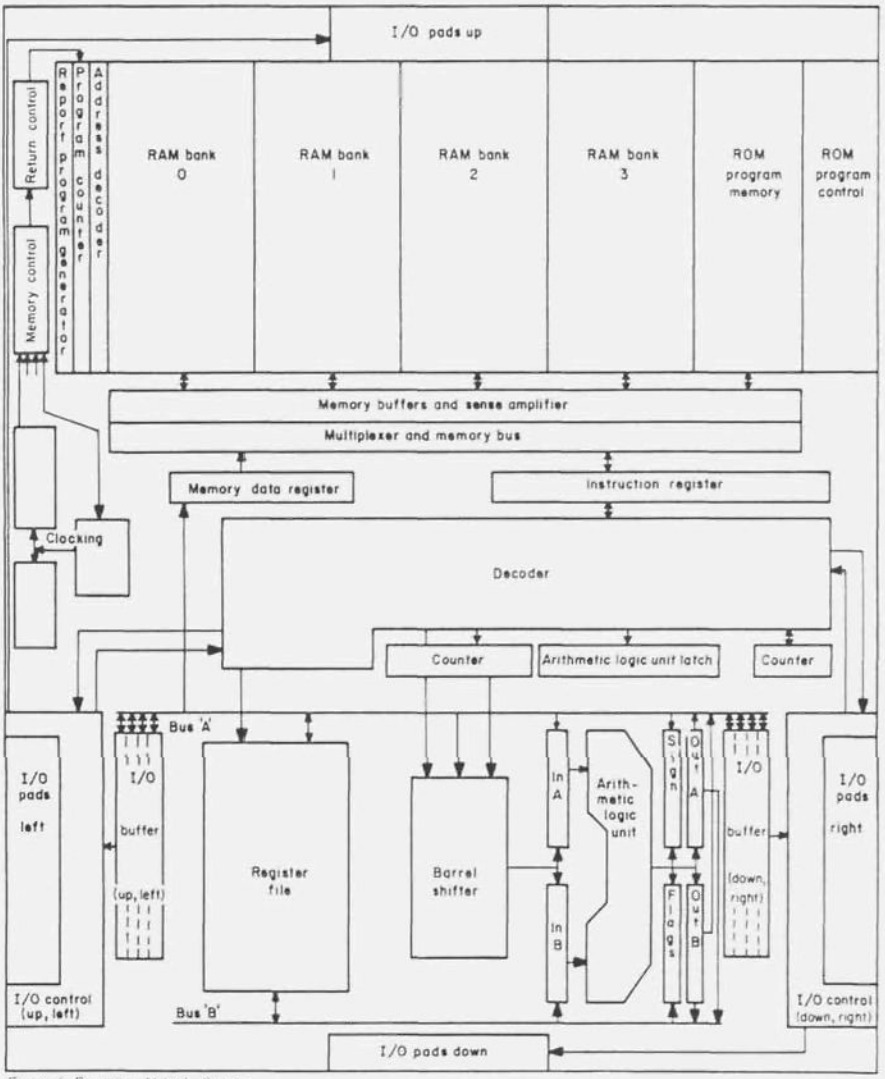

Figure 4. Functional block diagram

a preprocessor is needed. For a wavefront array the design of such a preprocessor is relatively easy since we do not have to consider the timing problems associated with the synchronous systolic array.

An example of an MDFL program for matrix multiplication on a wavefront array is given in Figure 3. This example clearly illustrates the simplicity of the resulting description. A complete list of the MDFL instruction repertoire as well as some more complicated examples and the detailed syntax have been given by Kung *et al*[12].

Wavefront architecture

The data-driven feature of the wavefront array is the key to get around the need for global synchronization – a potential barrier in the design of ultralarge-scale systems. In the wavefront architecture, the information transfer is by mutual agreement between a processor element and its immediate neighbours. Whenever the data is available, the transmitting processor element informs the receiver of the fact, and the receiver accepts the data when it needs it. It then conveys to the sender the information that the data has been used. This scheme can be implemented by means of a simple handshaking protocol[7,12]. The wavefront architecture can provide asynchronous waiting and consequently can cope with timing uncertainties, such as local clocking, random delay in communications and fluctuations in computing times[7,13].

The hardware of the processor element is designed to support MDFL[12]. Given the current state of the art in device technology, it is feasible to produce a single-chip wavefront processor. The main functional units to be considered are

- datapath, consisting of an arithmetic logic unit, a register file, a barrel shifter and some additional circuitry to facilitate CORDIC-type operations
- program memory, between 8 kbits and 32 kbits of dynamic RAM memory and a 1–2 kbit ROM memory
- simple control unit, consisting of a decoder, a program counter and some additional logic to allow program loading, conditional instructions etc.
- four asynchronous I/O interfaces, each with its own control unit

The functional block diagram of the resulting processor element is shown in Figure 4. On the basis of data available from recent VLSI chip designs[14,15] we made some estimates of the approximate size of the functional blocks. With reference to Figure 4, it should be possible to accommodate all of them on a chip of approximately 10 mm x 8 mm. The floorplan is very regular, because of the use of a large program memory, a bit-sliced structure for the datapath and a simplified control part.

Based on the numerical requirements of, for example, a matrix inversion a datapath width of 32 bits was selected as it seems to offer most in terms of applicability. The datapath itself is rather conventional, following the lines indicated by Mead and Conway[1] but with some additions to speed up multiplication and CORDIC-type operations. Floating point operations are possible by using two registers for each operand, one to represent the exponent and the other to represent the mantissa.

Applications of wavefront processing

Via the notion of wavefront processing, the data-driven computing scheme can be shown to be naturally suitable for all signal processing algorithms that possess recursivity and locality. The power and flexibility of the wavefront array and MDFL programming are best demonstrated by the broad range of application algorithms. Such algorithms can be roughly classified into three groups

- basic matrix operations such as
  - matrix multiplication
  - logical unit decomposition
  - logical unit decomposition with localized pivoting
  - Givens algorithm
  - back substitution
  - null-space solution
  - matrix inversion
  - eigenvalue decomposition
  - singular value decomposition
- special signal processing algorithms
  - Toeplitz system solver
  - linear convolution
  - recursive filtering
  - circular convolution filtering
  - digital Fourier transform
- other algorithms, eg
  - solution of partial difference equations
  - sorting

CONCLUSIONS

The rapid advances in VLSI device technology and design techniques have encouraged the realization of massively parallel array processors. We have stressed the importance of modularity, communication and system clocking in the design of VLSI arrays. For signal processing applications, as shown in the previous section, a large number of algorithms possess the properties of recursiveness and locality. These properties naturally led to the wavefront concept and to the use of an array of modular and locally interconnected processors as the computing medium. The wavefront array, as opposed to the systolic array, further stresses the features of asynchronous communications between processor elements and simple programmability by tracing the propagation of (computational) wavefronts. (Internally each processor element will be (locally) synchronized.) Both of these features are critical for a feasible and cost-effective design of future VLSI systems.

About the design of processor elements we remark that the reduction of the control part, based on the use of a microprogrammable processor element and a careful selection of the required instruction set, leads to a simple architecture and a regular floorplan. Moreover, by simplifying the instructions (ie decomposing complex instructions into sequences of simpler instructions), we are able to reduce the basic clock cycle and to gain a speed advantage.

Another feature of the wavefront array is that it can cope with variations in the interprocessor communication path delays. Hence it is more suitable for use in conjunction with flexible interconnection schemes. For example, the wavefront array seems a good candidate for waferscale integration, where the communication paths are not predictable because of rerouting of interconnections in order to get around faulty processor elements.

ACKNOWLEDGEMENTS

The authors wish to thank David Chang, W C Fang, David Lin, W K Lu and Govind Sharma of the University of Southern California for their contribution to the design of the wavefront processor element.

This research was supported in part by the US Office of Naval Research and the US National Science Foundation.

REFERENCES

1 **Mead, C and Conway, L** *Introduction to VLSI systems* Addison-Wesley, Reading, MA, USA (1980)

2 **Kinniment, D J** 'VLSI and machine architecture' in **Randell, B and Treleaven, P C** (eds) *VLSI architecture* Prentice-Hall, Englewood Cliffs, NJ, USA (1983) pp 24–33

3 **Anceau, F and Reis, R** 'Design strategy for VLSI' in **Randell, B and Treleaven, P C** (eds) *VLSI architecture* Prentice-Hall, Englewood Cliffs, NJ, USA (1983) pp 128–137

4 **Barbacci, M R** 'Instruction set processor specifications (ISPS): the notation and its applications' *IEEE Trans. Comput.* Vol 30 No 1 (1982) pp 24–40

5 **Lam, J** 'RTsim: a register transfer simulator' *Master's Thesis* Computer Science Department, California Institute of Technology (1983)

6 **Bryant, R E** 'An algorithm for MOS logic simulation' *Lambda* No 4 (1980) pp 46–53

7 **Kung, S Y and Gal-Ezer, R J** 'Synchronous vs. asynchronous computation in VLSI array processors' in *Society of Photo-Optical Instrumentation Engineers Conf.* (1982)

8 **Kung, S Y** 'VLSI array processor for signal processing' in *Conf. on Advanced Research in Integrated Circuits* Massachusetts Institute of Technology, Cambridge, MA, USA (1980)

9 **Kung, H T and Leiserson, C E** 'Systolic arrays (for VLSI)' in *Sparse Matrix Symp.* SIAM (1978) pp 256–282

10 **Kung, H T** 'Why systolic architectures' *IEEE Computer* Vol 15 No 1 (1982)

11 **Kung, S Y** 'From transversal filter to VLSI wavefront array' in *Proc. Int. Conf. on VLSI '83* (August 1983)

12 **Kung, S Y, Arun, K S, Gal-Ezer, R J and Bhaskar Rao, D V** 'Wavefront array processor: language, architecture and applications' *IEEE Trans. Comput.* Vol 31 No 11 (November 1982)

13 **Wann, D F and Franklin, M A** 'Asynchronous and clocked control structures for VLSI based interconnection networks' *IEEE Trans. Comput.* Vol 32 No 3 (March 1983)

14 **Fisher, A T et al.** 'Design of the PSC: a programmable systolic chip' in **Bryant, R** *(ed.) 3rd Caltech Conf. on VLSI* Computer Science Press (1983)

15 **Sequin, C H and Patterson, D A** 'Design and implementation of RISC I' in **Randell, B and Treleaven, P C** (eds) *VLSI architecture* Prentice-Hall, Englewood Cliffs, NJ, USA (1983)

Appendix B: Localization and Systolization of SFG's

Temporal Localization and Systolization of Signal Flow Graph (SFG) Computing Networks.†

S.Y. Kung, S.C. Lo
University of Southern California

J. Annevelink
Delft University of Technology

ABSTRACT

This paper addresses the theoretical and algorithmic issues related to optimal temporal localization(and systolization) of Signal Flow Graph(SFG) computing networks. Based on a cut-set localization procedure we propose an algorithm that computes the optimal localization of an SFG. The basic algorithm is then extended so that it can be used with hierarchically specified SFG's, thus significantly improving the computational efficiency. The algorithms can be easily coded and incorporated into a computer-aided-design (CAD) system.

† This research was supported in part by ZWO, the Dutch Foundation for pure scientific research, by the Office of Naval Research under contracts N00014-81-K0191 and N00014-83-C-0377 and by the National Science Foundation under Grant ECS-82-12479.

B.1 Introduction

In a recent paper we proposed to base the design and specification of Signal Processing Systems on the mathematical abstraction of the Signal Flow Graph [1]. The Signal Flow Graph (SFG) representation derives its power from the fact that the computations are assumed to be delay free, i.e. they take no time at all. Consequently, the need of tracing detailed time-space activities, as is usually done when specifying or verifying (systolic) array processors is avoided. Moreover, any delay present in the system has to be explicitly introduced in the form of so-called Delay branches. These Delay branches allow history sensitive systems to be described in a clear and unambigious way. Such a model with explicit delay (state) modeling, is consistent with the concern Backus expressed over an "extended" functional programming, e.g. the AST systems introduced in [2]. Although the abstraction provided by the SFG is very powerful, transformation of an SFG description to a wavefront or systolic array description, including the pipelining, can be made rather straightforwardly. Some existing theorems which may facilitate these transformations can be found in a recent paper by Kung [3]. The readers are referred to [4], [5], [6] and [7] for a review of several existing approaches.

In this paper we will introduce a set of algorithms that can be used to temporally localize an hierarchically specified SFG. Based on the algorithms discussed in the paper, our approach offers an effective starting point for the design automation and software/hardware techniques. This is because (1) SFG provides a powerful (although mathematical) abstraction to express parallelism, and yet (2) transforming from SFG to (the more realistic) systolic/wavefront arrays is straightforward.

B.2 Temporal Localization of an SFG

B.2.1 Signal Flow Graph

Signal Flow Graph's are probably the most popular graphical representation for scientific computation and signal processing algorithms. In this section we will assume that an SFG is given by a finite directed graph, $G=<V,E,d>$. The vertices V of the graph G, model the nodes. The directed

edges E of the graph model the interconnections between the nodes. Each edge e of E, connects an output-port of some node to an input port of some node, and is weighted with a delay count $d(e)$. The delay count is the number of delays along the connection. The vertices with either in-degree zero or out-degree zero are special ; they represent the **input** and **output** ports of the graph. Input and output ports are also referred to as **sources** and **sinks,** respectively.

To illustrate the power of the SFG notations let us now look at a parallel QR algorithm and its SFG representation. The QR algorithm transforms the initial matrix A in an upper triangular matrix R by means of an orthogonal transformation Q, Q A = R. The transformation is implemented by first applying the following decomposition,

$$Q^1 A = \begin{array}{c} x \; x \; x \; \ldots \; x \\ \hline 0 \\ 0 \qquad A^* \\ \vdots \\ 0 \end{array}$$

and, then repeatedly applying similar decompositions, namely Q^i, i = 2, 3, ..., N - 1.(Here N denotes the number of columns of A.) The above recursive algorithm can be mapped onto a parallel processing SFG network as shown in Figure B.1. It shows that after one recursion the submatrix A^* will be moved to the left and upwards. The delays "D" represent the state of the SFG, and contain the submatrix A* that will be processed in the next recursion.

B.2.2 Cut-Set Temporal Localization

In this section we will present the outline of an algorithm to temporally localize an SFG.

Definition Temporal Localization :

An SFG is **temporally localized** if there is at least one unit-time delay allotted to a data-processing node (and the corresponding data-transferring

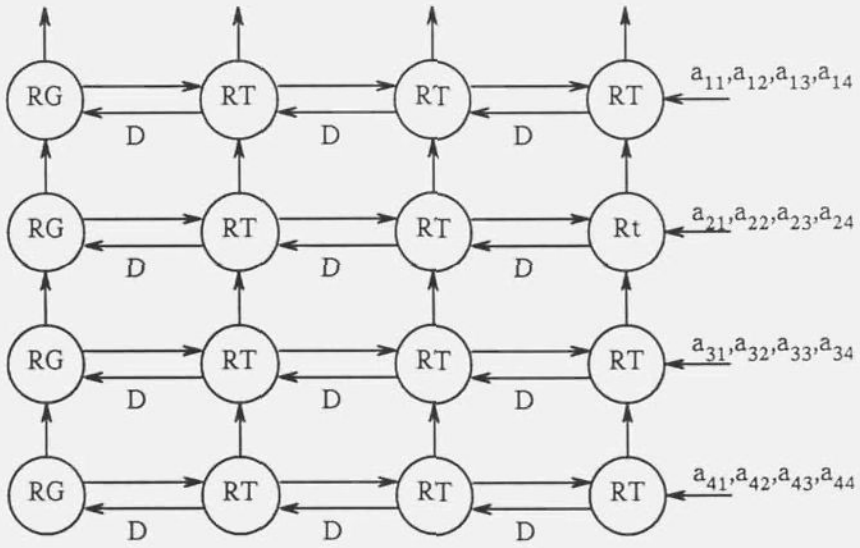

Figure B.1. An SFG example for the QR algorithm

edge), so that the signal transaction can be completed in the given time.

To temporally localize an SFG we have to transform the SFG into an **equivalent** SFG, in which there is at least one delay along every interconnection between two nodes. According to the definition given in [3], temporal locality is one of the characteristic properties of systolic arrays. In fact, converting a regular and locally interconnected SFG into a systolic array hinges upon the process of temporal localization.

The temporal localization algorithms proposed here are derived from the cut-set localization procedure introduced in [3].

Definition Cut-Set :

A cut-set in an SFG is a **minimal** set of edges which partitions the SFG into two disconnected components.

The localization procedure is based on two simple rules :

1. *Time-scaling* : All delays D may be scaled, i.e., $D \dashrightarrow \alpha * D$, by a single positive integer α. Correspondingly,the input and output rates also have to be scaled by a factor α (with respect to the new time unit D.

2. *Delay-Transfer* : Given any cut-set of the SFG, we can group the edges of the cut-set into **in-bound edges** and **out-bound edges** depending upon the directions assigned to the edges. Rule 2 allows advancing k D time-units on all the out-bound edges and delaying k time-units on the in-bound edges, and vice versa. It is clear that, for a (time-invariant) SFG, the general system behavior is not affected because the effects of lags and advances cancel each other in the overall timing. Note that the input-input and input-output timing relationships will also remain exactly the same only if they are located on the same side. Otherwise, they should be adjusted† by a lag of +k time-units or an advance of -k time-units.

We shall refer to these two basic rules as the (cut-set) **localization rules.** Based on these rules, we assert the following :

Theorem :

All computable‡ SFG's are temporally localizable.

Proof : We claim that the localization rules (1) and (2) can be used to "localize" any (targeted) zero-delay edge, i.e. convert it into a nonzero-delay edge. This is done by choosing a **"good"** cut-set and apply the rules upon it. A good cut-set including the target edge should not include any "bad edges", i.e. zero-delay edges in the opposite direction of the target edge. This means that the cut-set will include only (a) the target edge, (b) nonzero delay edges going in either direction, and (c) zero-delay edges going in the same

† If there is more than one cut-set involved, and if the input and output are separated by more than one cut-set, then such adjustment factors should be accumulated.

‡ An SFG is meaningful only when it is computable, i.e., there exists no zero-delay loop in the SFG.

direction. Then, according to Rule (2), the nonzero delays of the opposite-direction edges, the "source" edges, can "give" one or more spare delays to the target edge (in order to localize it). If there are no spare delays to give away, simply scale all delays in the SFG according to Rule (1) to create enough delays for the transfer needed.

Therefore, the only thing left to prove is that such a "good" cut-set always exists. For this, we refer to Figure B.2, in which we have kept only all of the zero-delay *successor* edges and the zero-delay *predecessor* edges connected to the target edge, and removed all the other edges from the graph. In other words, Figure B.2 depicts the bad edges which should not be included in the cut-set. As shown by the dashed lines in Figure B.2, there must be "openings" between these two sets of bad edges — otherwise, some set of zero-delay edges would form a zero-delay loop, and the SFG would not be computable. Obviously, any cut-set "cutting" through the openings is a "good" cut-set, thus the existence proof is completed. In graph theory this result is known as the **colored arc lemma.** It is clear that repeatedly applying the localization rule (2) (and (1), if necessary) on the cut-sets will eventually lead to a temporally localized SFG.

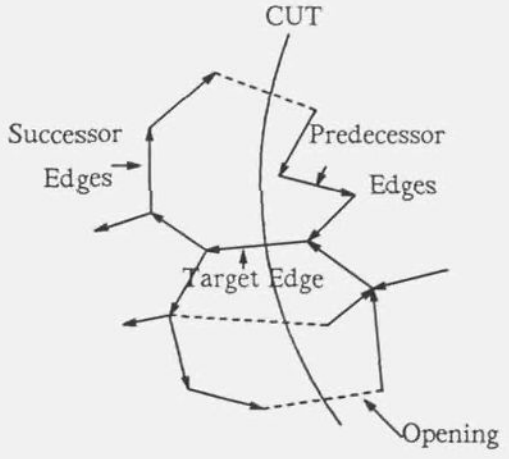

Figure B.2. Openings between bad edges ensure the existence of a good cut-set

As an example, Figure B.3 shows a localized(and systolized) version of the QR SFG given in Figure B.1. According to the definition in [3], Figure B.3 in effect represents a systolic array configuration for the QR algorithm.

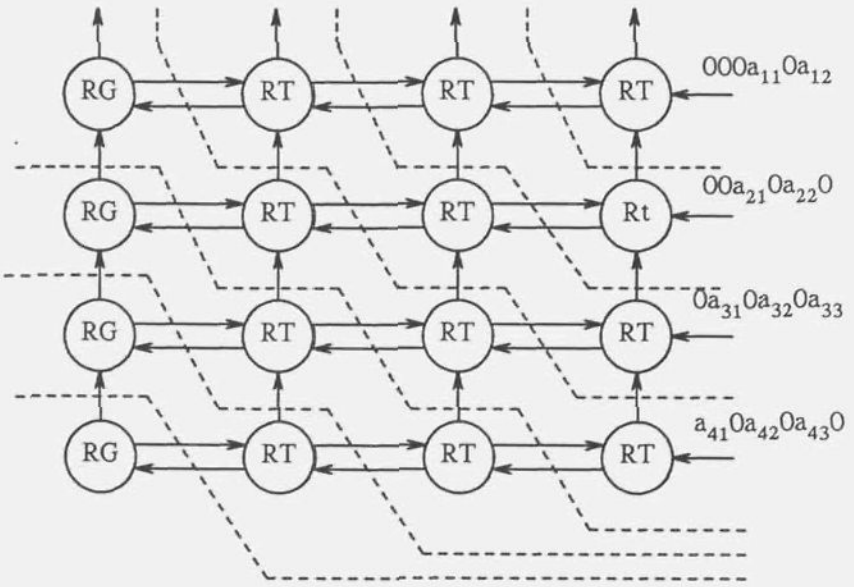

Figure B.3. Systolized SFG for the QR algorithm

B.2.3 Optimal Time Scaling and Temporal Localization

We have so far discussed the basic theorem asserting the temporal localizability of SFG's. Now let us address the important question of optimal time-scaling. Since the throughput rate of the computing network is inversely proportional to the scaling factor α, it is obvious that the optimal α will be the minimum integer needed to complete the localization procedure.

To ensure optimality of the cut-set procedure, **the only modification we need is to confine the application of the delay-transfer operation to a restrictive class of good cut-sets, namely, Non-Rescaling** (NR) cut-sets. Here, a Non-Rescaling(NR) cut-set is a good cut-set in which all the edges (excluding input edges†) in the opposite direction to the target edge have at

least two (instead of one) delays. Therefore, the (optimal) cut-set procedure is just a simple modification of the cut-set localization rules in section B.2. Again, it has two simple guidelines:

1. *Delay-transfer* - Once an NR cut is determined, we can simply apply the delay-transfer operation along the cut and localized the target edge(s). It is optimal in the sense that no additional time-scaling will be needed .

2. *time-scaling* - If there exist no NR cut-set, it implies that the current rate is too fast and extra slow down will be needed. (A formal proof will be given in a moment.) Therefore, we increment α (each time by one) until a NR cut-set can be identified.

Theorem :

The above procedure suffices to convert an SFG network into a localized network with optimal throughput rate,(i.e. minimum α).

Proof : To prove the theorem, we need only to show that if an NR cut does not exist then an increment of the scaling factor will be necessary. For a given target edge if there exists no NR cuts, then according to the Colored Arc Lemma, there exists a loop containing the target edge and yet containing no edges with more than one delay. Completing the computation around the loop (so that the resultant be available for the next operation at the beginning node), it will take as many time-units as the number of edges in the loop. It means that the time-delay assigned will be short by at least one time-interval due to the zero- delay in the target edge. Consequently, a rescaling of α is apparently needed.

By the computability of the SFG, the loop should also contain at least one delay. Therefore, a proper time rescaling will suffice to settle the timing problem of the loop. Consequently an NR cut should now be available (Note

---

† An input edge is an edge directly connected to an input node.

that, according to the Colored Arc Lemma, if there are no more "bad" loops containing the target edge, then an NR cut should now exist.).

B.2.4 Optimal Cut-set Localization Algorithm

The theorem suggests that the algorithm should search an NR cut. When such cut does not exist, then a loop containing the target edge will be formed. We can take a proper action depending whichever comes first. To accomplish this, we introduce a notion of supernode - a clustering of nodes according to the following search procedure :

Start with a target edge and expand the supernode by tracing all the "bad" predecessor edges (similar to what is shown in Figure B.2) until

1. either the supernode is surrounded by the eligible edges only. Then these edges form an NR cut-set. (Action: Apply the proper delay transfer.)

2. or the supernode cluster ends up to the terminating vertex of the target edge. (Action: apply a proper time rescaling, and restart the NR cut-set search procedure for the same target edge.) Note that after time-rescaling the source delay edges will become eligible for the next cut-set selection.

For a pictorial example, corresponding to Case (2), it is possible that the search procedure ends up with a loop just like what is shown in the upper loop in Figure B.2. On the other hand, corresponding to Case (1), a possible supernode (and NR cut set) will be the one corresponding to the graph encircled by the cut (solid line) as shown in Figure B.2.

The above rules are the basis of the optimal cut-set procedure given in Appendix B.1. Due to its simplicity, the algorithm may become preferable when the network, or all the loops in the network, are of small scale. The algorithm given in Appendix B.1 will localize the target edge as well as the potential target edges in the cut set surrounding the supernode.

For an efficient algorithm the α will be computed according to the delay distribution along the loop, (instead of being incremented only by one each time.) Another potential improvement is to localize the potential target

edges inside the supernode in addition to those surrounding it. Both of these improvements are incorporated in the algorithm listed in Appendix B.2.

B.3 Hierarchical SFG's and HIFI Design Methodology

In the HIFI design environment [1] a system is (graphically) represented with a Signal Flow Graph (SFG), thereby simplifying the space time description. The HIFI design method is based on the concept of **node refinement.** Starting from a single node specification the system is specified in more and more detail by applying decomposition functions to the nodes. By naming nodes or functions performed by the nodes, the design is (hierarchically) decomposed, allowing the designer to focus his attention on the specific node or function selected. A node refinement is used to specify both **structural** and **behavioral** refinement. Based on algorithm analysis, e.g. the recursive decomposition scheme, we will be able to identify certain useful and often occurring structures, e.g. arrays, trees etc.

In the previous section we have discussed an algorithm to localize a SFG specified as a directed graph $G = <V, E, d>$, where $d(e)$ represents the number of delays on an edge e of E. This algorithm, although it computes a localization of G with an optimal throughput rate, becomes inefficient when we have to localize SFG's consisting of hundreds or thousands of nodes. Although the actual complexity of the algorithms is difficult to determine precisely, it seems clear that there is no obvious way to reduce the complexity. Therefore, in this section we will derive an alternative algorithm. This algorithm will be much more efficient, because it takes advantage of the regularity of an SFG. For a regular SFG, and an SFG needs to be regular in order to be implementable as a systolic array, the complexity is no longer determined by the number of nodes in the SFG as a whole, but instead by the number of nodes in the largest "node refinement". The actual number of nodes in a "node refinement" is small, usually not more then 10 - 15. The term "node refinement" was introduced in [1]. For now it suffices to say that a "node refinement" specifies the replacement of a node by a graph (cf. sect. A3.3). Node refinements can be used to specify an SFG, starting from a single node and ending up with the complete graph, by successively applying node refinements to the graph defined sofar. The

specification of an SFG by node refinements is pictorially represented in Figure B.4.

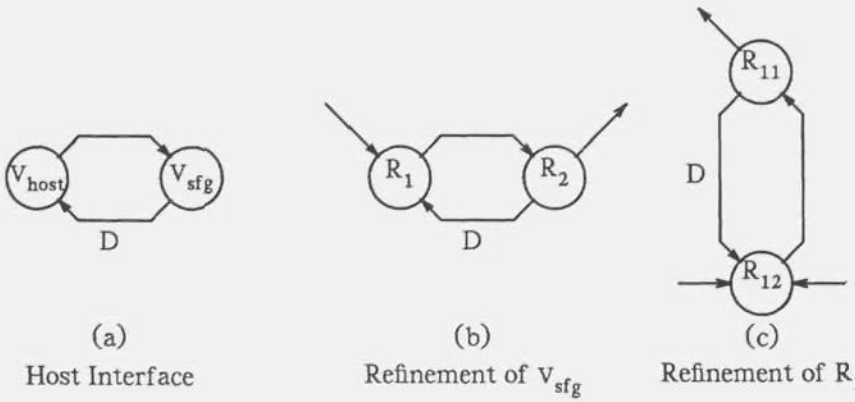

Figure B.4. Hierarchical Specification of an SFG

B.3.1 SFG Model

We assume that an SFG is given by a tuple $<Gt, R>$, where Gt is a graph denoting the SFG and its interface to the host processor, and where R is a set of node refinements.

The graph Gt is very simple. It consists of two vertices. One, *Vhost* represents the host processor, the other, *Vsfg* represents the abstraction of the first (top) refinement of the SFG. The connections between these two vertices represent the I/O connections between the SFG and the host processor (cf. Figure B.4(a)). A node refinement is modeled by a tuple $<Vh, Gr>$ where Vh represents the (hierarchical) abstraction of the refinement. Gr represents the implementation of the node refinement. Gr is modeled by a finite directed graph, $Gr = <V, E, d>$, like any of the SFG's discussed in the previous section. The vertices V of the graph Gr, model the nodes introduced by the refinement. Note that the nodes can be further refined, and that, implicit in the specification of the refinements, there is a partial ordering '$<$' relating the node refinements to each other. Vh, the abstraction of a node refinement, is a vertex with input and output ports. The input and output ports of Vh correspond to the input and output nodes

of *Gr*. For an hierarchical specification of SFG's this model would be sufficient. It allows us to specify the replacement of a graph by its abstraction and vice versa.

In order to define an hierarchical localization algorithm, we have to extend the model given above in two ways. First we define a lag number for every input and output port of the vertex *Vh*. The lag numbers are introduced, because if we localize the refinement graph *Gr* of a node and then replace this graph by its abstraction *Vh*, the resulting (partly localized) graph will no longer be a graph with zero-delay nodes, that is localized when all edges carry at least one delay. Instead, the delay required on an edge from an input to an output port will be given by the difference of the lag numbers of these ports.
In order to determine the optimal throughput rate of a graph *Gr*, we need to know the optimal throughput rates of the nodes that are further refined. Therefore, the second extension is the introduction of the minimal slow-down factor α of *Vh*.

To simplify the specification of the localization algorithms we model all vertices occurring in a *Gr* graph as described above. A node that has no further refinements, will be denoted by a vertex v, that has a lag of 1 for all of its input ports, and zero for its output ports. The slow down factor of the vertex will be 1. We also introduce a function *lag(edge)*, that returns the lag number associated with the edge. The lag number of an edge is the difference of the lag numbers of the ports that the edge connects to in the terminating and initial vertex of the edge, respectively. The lag number denotes the number of delays required to localize the edge.

B.3.2 Algorithm
To localize a graph given by a set of node refinements we have to localize the node refinements. For this we can use the localization algorithms given in the previous section. However, we need to extend these algorithms such that they can handle the more general model of a vertex given above. This extension is fairly simple, the complete algorithm is given in Appendix B.2. Note that in order to localize an edge in a graph containing vertices of the type discussed above, the number of (local) delays must be greater or equal

then the lag number of the edge, i.e. d'(edge) >= lag(edge).
A graph *Gr*, denoting a node refinement, can be localized only when we know the abstractions *Vh* of all the nodes in *Gr*. For nodes that are not further refined this abstraction is known. However, to determine the abstraction of a node that is further refined, we will have to localize that refinement first. It follows that we have to localize the graphs *Gr* starting with those refinements that contain only nodes whose abstraction is already known. After localizing all node refinements, we refine the graph *Gt*, denoting the interface with the host processor. Once this graph is also localized we essentially know the localization of the entire SFG. The only thing left to do is to update the local delay counts of the node refinements in order to take care of differences between the slow down factor of the graph *Gt* and the slow down factors of the node refinements. The procedure HLoc_graph(), given below, computes the localization of all the node refinements and the graph *Gt*. To derive a localized version of the entire SFG, we simply replace the nodes by their (localized) refinements.

B.4 Conclusions

This paper addressed the theoretical and algorithmic issues related to optimal temporal localization(and systolization) of Signal Flow Graph(SFG) computing networks. Based on a cut-set localization procedure we proposed an algorithm that computes the optimal localization of an SFG. The basic algorithm is then extended so that it can be used with hierarchically specified SFG's, thus significantly improving the computational efficiency. The algorithms can be easily coded and incorporated into a computer-aided-design (CAD) system, however, we still need to do more work on the hierarchical part, especially related with the regular structures such as arrays.

Thinking in retrospective, we note that the major effort on the optimal time rescheduling is on the rescaling and the redistribution of Source Delay, D. In any dynamic circuit, initial-condition-data are always assigned for all the delay elements. Thus, the actual data at the output of D has a net time allowance of α time-units, i.e. the time needed for the consumption of the initial state. On the other hand, the time saved by the consumption of initial

```
Procedure HLoc_Graph()
begin
  N_Loc_R := R, the set of all refinements ;
  while ( N_Loc_R is not empty) do
  begin
    Nr := "smallest" node refinement in N_Loc_R ;
    Loc_graph(Nr.Gr) ;
    determine abstraction Vh of Gr ;
    delete Nr from N_Loc_R ;
  end
  Loc_graph(Gt) ;
  α := slow_down factor of Gt ;
  forall (refinements Nr in R) do
    if (Nr.α < α) then
      update local delay count of Nr.Gr ;
end
Note : The "smallest" node refinement is selected using
       the partial ordering relating the refinements.
```

data tokens may be most easily computed by self-timed handshaking[3]. By the same account, the "pure delay" addition will become unnecessary, because the operation firing will be self-timed. Therefore, it is probably fair to say that there will be an optimal compromise between the synchronous processing (i.e. systolic array) and the asynchronous processing (i.e. wavefront array). For example, it should be possible to recognize a locally-clustered sub-network, while the entire network is (globally) asynchronous. The identification of the locally clustered parts should be easy given an hierarchically designed SFG. When the compromise is reached, then, our proposed procedures can be used for the local sub-network systolization, while the burden of global synchronization should be replaced by self-timed handshaking.

References

[1] S.Y. Kung, J. Annevelink and P. Dewilde, "Hierarchical Iterative Flowgraph Integration for VLSI Array Processors" In : Proc. USC Workshop on VLSI and Modern Signal Processing, Los Angeles, Ca. Nov. 1984

[2] Backus, J., "Can Programming Be Liberated from the Von Neumann Style ? A Function Style and Its Algebra of Programs" Comm. ACM 21, 613 - 641, Aug. 1978

[3] S.Y. Kung,"On Supercomputing with Systolic/Wavefront Array Processors", Proc. IEEE, Vol. 72, No. 7, July 1984, pp. 867 - 884.

[4] C.E. Leiserson, F.M. Rose and J.B. Saxe, "Optimizing Synchronous Circuitry by Retiming" in Proc. Caltech VLSI Conf. (Pasadena, CA), 1983

[5] A. Fettweis, "Realizability of Digital Filter Networks" AEU, Band 30 1976, Heft 2, pp. 90 - 96.

[6] C. Caraiscos and B. Liu "From Digital Filter Flow Graphs to Systolic Arrays" To appear, IEEE Trans. Acoust. Speech, Signal Processing, 1985

[7] J.V. Jagadish et al., "On Hardware Description from Block Diagrams" in Proc. IEEE ICASSP (San Diego, CA) 1984

Appendix - B.1

Note : $d(e)$: delay count in original SFG
$d'(e)$: delay count in localized SFG

Procedure *CutLoc()*
begin
$\alpha := 1$;

```
  T := set_of target_edges ;
       [* Input edges are excluded from target-edges *]
  while (T isnot empty) do
  loop
    Target_edge := e, where e belongs to T ;
    cut_found := true ;
    good_edges := <> ;
    GoodCut( <>, Target_edge) ;
    if (cut_found is true) then
    begin [* a good-cut is found, do delay transfer *]
      d'(Target_edge) := 1 ;
       delete Target_edge from T ;
      forall (edges e of good_edges ) do
          if (e is in direction of Target_edge) then
           if (e is in T) then delete e from T ;
           d'(e) := d'(e) + 1 ;
         else d'(e) := d'(e) - 1 ;
    end
  end_loop
end

Procedure GoodCut(SuperNode, T_edge)
begin
  T_vertex := T_edge.initial_vertex ;
  if (T_vertex is in Supernode) then
  begin
    if (Target_edge is T_edge) then
    begin [* if target edge is in loop *]
      cut_found := false ;
       α := α + 1 ;
       forall (e in E) do
         d'(e) := d'(e) + d(e) ;
    end
    else
```

```
        good_edges := good_edges - T_edge ;
    end
    else if (T_vertex is an input node) then
      [* Include T_edge into good_edges. Edges connected to input nodes
         can have negative delays indicating the number of delays needed. *]
      good_edges := good_edges + T_edge ;
    else
    begin
      [* Find a good cut by looking at the edges incident on T_vertex.
         Include all eligible edges into good_edges. If there are bad edges
         (edges incident into T_vertex with 1 or 0 delays), expand the supernode
         to include all initial vertices of bad edges and try to find a good
         cut recursively. *]

      SuperNode := SuperNode + T_vertex ;
      good_edges := good_edges + all good edges of T_vertex ;
      forall ("bad" edges incident into T_vertex) do
        GoodCut(SuperNode, bad_edge) ;
    end
end
```

Appendix - B.2

Note : *md_cnt* : missing delay count
td_cnt : transfer delay count
sd_cnt : source delay count
lag() : lag number of an edge
The "target-edges" are those edges for which lag(edge) > d'(edge)

```
Procedure LocGraph(NodeRef)
begin
  α := maximum of slow-down factors of vertices in
         the graph Gr of the NodeRef ;
  Initialize localized delay count ;
```

```
  T := set_of target_edges ;
  while (T isnot empty) do
  loop [* Localize edge *]
    Target_Edge := e, such that e in T ;
    LocPath( <>, Target_Edge, lag(e) - d'(e), d(e), loop_fnd) ;
    delete Target_Edge from T ;
  end_loop
end

Procedure LocPath(SuperNode, Target_Edge, md_cnt, sd_cnt, loop_fnd)
begin [* Transfer at least md_cnt delays to the Target_Edge. *]

  T_vertex := Target_Edge.initial_vertex ;
  if (T_vertex is in SuperNode) then
  begin [* Found a loop, compute α and set loop flag *]

    α_loop := [(md_cnt + α * sd_cnt ) / sd_cnt] ;
    if (α < α_loop) then
    begin [* Update α and increment local delay counts d'(.) *]
       forall (e in E) do
      begin
         d'(e) := d'(e) + α_loop - α * d(e) ;
         if (lag(e) <= d'(e)) then
           delete e from T ;
      end
        α := α_loop ;
    end
    T_vertex.loop_begin := loop_fnd := true ;
  end
  else if (T_vertex is an input node) then
  begin
    d'(Target_Edge) := d'(Target_Edge) + md_cnt ;
    add md_cnt to lag number of input portof hTarget_Edge ;
  end
```

```
else
begin
  [* Find a good cut by looking at the edges incident
     into T_vertex. If these edges do not make a good cut,
     then transfer the appropriate number of delays to these
     edges by a recursive call to LocPath, with md_cnt set to
     the required number of delays. If LocPath finds a loop, it
     will set the loop_fnd flag. In that case we may have to
     transfer more then md_cnt delays. The number of delays
     that must be transferred is denoted by td_cnt. This variable
     is initially set to md_cnt, and updated for every loop found
     by LocPath. In case LocPath finds a loop we want to transfer
     the maximum possible number of delays. *]

  td_cnt := md_cnt ;
  T_vertex.md_cnt := md_cnt ;
  T_vertex.sd_cnt := sd_cnt ;
  T_vertex.loop_begin := false ;
  SuperNode := SuperNode + T_vertex ;
  forall ( edges e1 incident into T_vertex) do
    if (td_cnt + lag(e1) > d'(e1)) then
     begin [* It's a "bad" edge *]
       loop := false ;
       LocPath(SuperNode, e1, td_cnt + lag(e1) - d'(e1),
                              sd_cnt + d(e1), loop) ;
      if (loop is true) then
         if (td_cnt + lag(e1) < d'(e1)) then
         begin [* Update td_cnt *]
            td_cnt := d'(e1) - 1 ;
            loop_fnd := true ;
         end
     end
  if (td_cnt > md_cnt) then
  [* LocPath found a loop, and wants to transfer more then the requested
```

```
        md_cnt delays. Make sure that every edge can provide the td_cnt delays.
        If not, then call LocPath again, to provide the additional delays on
        the edge *]
       forall ( edges e1 incident into T_vertex) do
         if (td_cnt + 1 > d'(e1)) then
           LocPath(SuperNode, e1, td_cnt + lag(e1) - d'(e1),
                                  sd_cnt + d(bad_edge), loop_fnd) ;
    end
    if (loop_fnd is true ) then
       if( T_vertex.loop_begin is true ) then
        loop_fnd := false ;
    [* Now for every edge e emergent out of T_vertex, we have
      d'(e) > td_cnt + 1, so that we can transfer td_cnt delays
       from the edges incident into T_vertex to the edges incident
       out T_vertex. *]
    forall (edges e1 incident out T_vertex) do
    begin [* Update delay count *]
      d'(e1) := d'(e1) + td_cnt ;
       if (e1 is in T) then delete e1 from T ;
    end
    forall (edges e1 incident into T_vertex) do
    begin [* Update delay count *]
      d'(e1) := d'(e1) - td_cnt ;
       if (e1 is in T) then delete e1 from T ;
    end
  end
end
```

Appendix C: Object Oriented Data Management

Object Oriented Data Management Based on Abstract Data-types†

J. Annevelink and P. Dewilde

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. This in turn poses problems of design data management which are considered here. We show that an object oriented data management system is a step towards solving the problems. The method proposed here is based on a systematic specification of data structures and data access methods, using abstract data-type specifications. As a result, the data management is completely transparent to the application programs. The data-management system is written in Enhanced C (EC), a set-oriented extension of the language C.

Keywords: object oriented, data-management, abstract data types, sets

† This research was supported in part by the commission of the EC under ESPRIT contract 991

C.1 Introduction

An important problem that arises in the development of (VLSI) design tools is that of the management of the design data. One of the problems is that what is easily referred to as "design data", usually consists of many different types of information, and that there exist complex relationships between different types of information. Moreover, depending on the type of information, efficient methods to *access*, *modify* or *update* the information differ considerably. This motivates an object-oriented approach, which in turn calls for a data-management system that is able to represent and manipulate objects as single entities, taking care of and maintaining the relations that may exist between the components of an object.

In this paper we present a simple object oriented data-management system that can be efficiently integrated into a variety of (VLSI) design tools, and that allows the construction of an efficient and powerful user interface layer for controlling the design tools, and for interrogating the status of the design system. The system as presented here is oriented towards the application programmer, in the sense that we have only a programming language interface to the database. Forthcoming work will describe an object-oriented user interface on top of the existing system.

C.1.1 Related work

Much of the work reported in the literature about data management for VLSI design, or for CAD in general, describes management of design information as collections of raw data in files. The management of the data in the design files is left to one or more design tools. The data management system must manipulate the design files on the basis of whatever additional information is available regarding the contents of the files. Often this information is encoded by choosing meaningful names for design files and grouping related design files together e.g. in a directory. It is then possible to make an interface that, based on a more or less complex model of the design process, relieves the designer from remembering all conventions, concerning where the design files must be stored. For example, the data management system described in [McLe85] provides the user with an interface layer, the chip manager, that handles the mapping between (cell)

names (provided by the designer), and actual file names. In this system, all design data is stored in the form of cells, which have a *cellname* given by the designer. Cells also have a *celltype*, which indicates the kind of data they contain. Cells with the same name, but that differ in type, are regarded as different aspects of the same part of the design. A *cellname* and a *celltype* are the only attributes that a cell itself has, as far as the chip manager is concerned. The contents of a cell are known only to the application programs, which can get access to the contents of a cell via the chip manager.

Other approaches use the relational database model [Date81]. The advantage of the relational model is that it can model all data in a simple and uniform way, i.e. as *relations*. A *relation* is usually represented in storage as a table, where the rows represent tuples and the columns represent attributes. The primary disadvantage of the relational model is its limited abstraction capability. Relations that can not be modeled directly must be normalized to relations that can, whereby a potentially meaningful relationship must sometimes be broken down into two or more relations whose meaning can not be directly understood. It is also difficult for a designer to manipulate the data in the database, since the conceptual model of the designer does not match with the way the data is represented. Moreover, in order to retrieve or update a single piece of information it is often necessary to access several relations. As a result of the size of a VLSI design the efficiency is usually low, and the resulting system too slow for practical use. A further problem associated with the relational data model is the fact that it is actually necessary to express all relationships between design objects, down to the lowest levels of detail. The introduction of design abstractions, e.g. the *cell* in the system mentioned above, is complicated by the fact that most relational database systems allow only a very limited set of data-types for specifying the attributes of a relation. New, so-called semantic database models try to remedy the disadvantages of the relational database model (cf. e.g [Bekk83, Afsa84])..

Other approaches look more specifically at the (VLSI) design environment. For example, the data model developed in [Bato85] incorporates a number

of concepts specific to a design system, e.g. objects have an interface description and an implementation description. Also, versions of objects and instantiation, i.e. the replication of a generic object, are captured by the model. An overview of the characteristics of the VLSI design environment and the information management requirements it poses is given in [McLe83].

File oriented data management systems do not provide the methods needed to access the information contained in the design files. However, since the application programs will have to access this information, the application programmer is left with the following two problems :

1. *How is the structure of the design files specified* ?
2. *How are the methods to access the design files specified* ?

Because these problems have not been solved systematically, file oriented database systems are potentially costly to maintain and evolve. Adding or modifying design tools requires the data-formats of the files to be known as well. This problem is complicated by the fact that the actual specification of the file data-formats is usually buried deep inside the design tools. Also, the design of a uniform and easy to use user interface is hampered by the fact that each file usually has its own data-format. At the positive side this very property can make a file oriented database system very efficient to use for the design tools, since data-formats can be optimized to the application at hand.

In this paper we will concentrate on two aspects of the problem sketched above.

1. A general and systematic method to specify and implement the data structures and access methods required to represent and manipulate (VLSI) design data.
2. The definition of a simple, low-level interface library containing a set of routines by which data can be efficiently transferred to and from secondary storage.

In our view, the key to a general and systematic method lies in the separation of the data structure specification from the design and implementation of the application programs. In particular, the HIFI [Anne86] design tools are being implemented this way. By separating the data structure specification from the design of the application programs, different tools can use the same data structure by simply including the files containing the data structure specifications. Moreover, the data-structures need to be specified only once, ensuring the consistency of different application programs.

The structure of this report is as follows. In section two we describe the ideas behind the new method. In section three we give an example of an object definition, and show how it is used in an actual application program. In particular, we will show the way the actual data management is made transparent to the applications programmer, as well as how to use and construct sets and sequences of objects. In section four, we will discuss the specification of objects in more detail. In section five, we will briefly describe the data-types required to implement the data-management method. There we will also discuss the low-level interface library that is used to efficiently access objects in secondary storage. Finally, in section six, we will draw a number of conclusions, based on the results derived so far.

C.2 Basic Philosophy

In this section we describe the main ideas behind the design and implementation of an object oriented data management package (DMP). The DMP is implemented in the form of a set of parameterized data-type definitions, and was initially developed for the HIFI design system [Anne86].

It is written in **Enhanced C** (EC) [Katz83], a high-level set-oriented extension of the C-language [Kern78]. EC allows a programmer to specify new data-types, using the concept of a **cluster**. A cluster can be viewed as a parameterized data-type specification. It can be mapped to a specific data-type by specifying values for the cluster parameters. The power of EC derives in part from the fact that it is possible to give types (i.e. type

names) as parameters. In mapping a cluster to a data-type, the programmer specifies an implementation of the type. The operations that can be applied to the objects (or values) of the new type are specified as part of the cluster definition. For more information on EC the reader is referred to [Katz83a, Katz85].

The data management package is essentially a set of cluster definitions. These cluster definitions are available to the applications programmer to create new data-types. The point here is that the actual data management, i.e. *when and how* to read, respectively write an object from and to secondary storage, is transparent to the applications programmer. This is dealt with via the operations defined by the clusters provided to the applications programmer. In effect, the EC compiler is used to *generate* the procedures necessary to access the information stored in the database. In order to do so, the compiler uses the type information supplied by the application programmer.

C.2.1 High Level Specification of Design Data

A DMP database has several conceptual layers. The first layer is the "design tool layer". The application programmer, when writing a design tool, will have to define the data-structures used to store the input and output data needed by the tool on secondary storage. We will require the application programmer to do so by defining **data-types**. Instances of these data-types can then be manipulated using a set of standard operations. The implementation of these operations depends on the type. However, they can be generated automatically from the definition of the data-type itself. The code written by a programmer can thus be separated into:

1. Data-type definitions
2. Application code

In general, a data-type is used to define a class of objects sharing a set of attributes. The attributes may denote either properties of the object being modeled by the data-type, e.g. its position, its name etc., or relations of the object with other objects, e.g. the relation between a transistor and the cell

it is part of, or the relation between a cell and the set of its input or output terminals.

The implementation of the data-types defined by the application programmer is handled by the second layer. Here we translate each data-type definition to an EC cluster definition.

The third layer defines the routines that actually access the database.

An overview of the steps required to construct a design tool is given in Figure C.1.
As can be seen in Figure C.1 the application programmer has to define a *Data-type Definition* and *Application Code*. The *Data-type Definition* is first mapped to an EC cluster using a procedure labeled *Data-type Map*, then (separately) compiled and added to a library of access functions. The *Application Code* is also compiled. Here we include the cluster definitions in the code, so as to enable the EC compiler to generate appropriate code, depending on the type of the variables and the operations performed on them. The access functions are added to an Access Function Library, and use elementary fetch and store functions from a Database Library (cf. section C.5). The last step then links the object code with the functions in the libraries to produce an executable program (*Design Tool*). From Figure C.1 we can see that we need to define the data-types only once. Subsequently all Application Functions simply include the corresponding cluster definition.

The advantages of having the data-type specified explicitly are:

1. The application programmer does not have to define routines for accessing and modifying the data on secondary storage.
2. Communication between design tools is easily accomplished, since they can use each others input and output data structures.
3. The consistency of the design data is maintained by the access routines.
4. Documentation and maintenance of design tools is simplified.

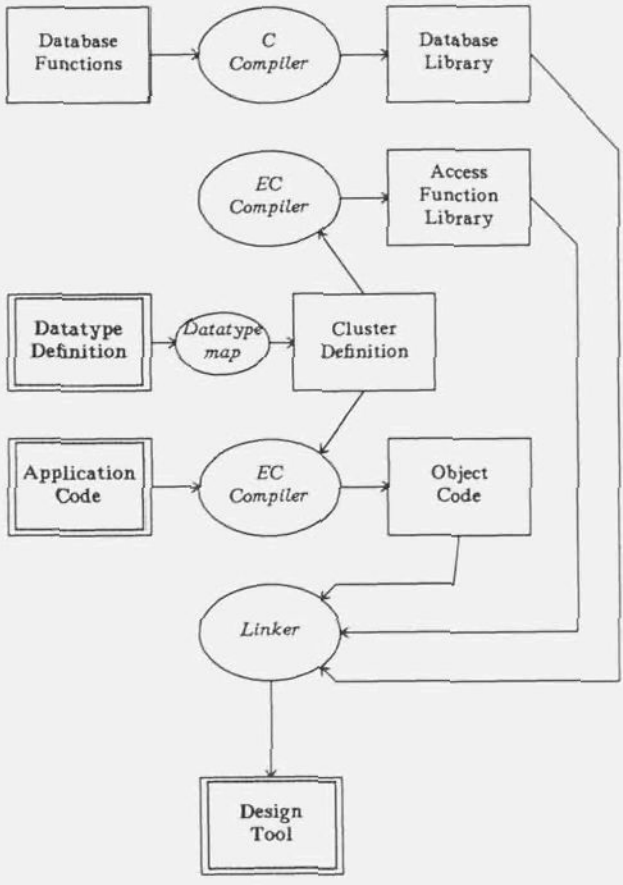

Figure C.1. Overview of the programming environment

The price we have to pay for these advantages is a slight reduction of efficiency. However, since the application programmer can define his own datatypes, the amount of overhead is controllable. It is possible for example to use the datamanagement system to get access to files, whose contents are then considered a black-box for the data management system.

C.2.1.1 The design-tool layer

A DMP database consists of a large collection of objects, each of which is an instance of a particular type. Each object has a unique key that is used both as a symbolic identifier for inter object references, and to retrieve the object from the database. The objects are stored in one or more disk files†.

The datamanagement method is based on a systematic definition of the object types. Given these systematic definitions, which amount to the definition of a dataschema, it is possible to define access procedures, that enforce certain consistency constraints. The metaschema of a database built using the DMP package is shown in Figure C.2.

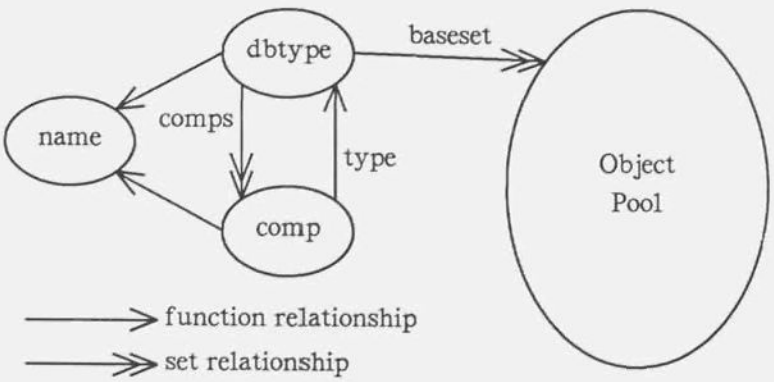

Figure C.2. Metaschema of a DMP Database

The metaschema shows that a DMP database is a collection of types. Every type has a name, a set of components, and a baseset. The baseset of a type contains references to objects that are instances of the type, and that are *visible* in the database. All objects in the database are stored in the so-called

† The detailed implementation depends on the structure of the DMP database library. (cf. section C.5.1)

object pool. This object pool can contain instances of a type that are not in the baseset associated with the type. These objects are *invisible*, and can be accessed only via objects that are visible in the database.

The principal abstraction implemented by a type is aggregation. An object of a particular type is an aggregation of component objects of (simpler) types. This is depicted by the (double headed) arrow pointing from *dbtype* to *comp* in Figure C.2. The arrow from *comp* back to *dbtype* denotes the type of the component. Depending on the properties of objects modeled by a type, the application programmer can choose the most appropriate abstraction for the components. A type that defines an object as an aggregation of *simpler* objects is called a tuple type. In addition to tuple types we can have set, sequence and reference types. For example, the baseset of a type is a set of references to objects of the type. Set, sequence and reference types are derived from the definitions of the tuple types, i.e. the application programmer is required to give only the definitions of the tuple types. The syntax† for the definition of a tuple object type is as follows:

---

† To describe the syntax of our specifications, we will use the meta language proposed by Wirth [Wirt77].
This language has two types of symbols:

— *terminal symbols* : These symbols must be present literally. They are denoted by words between double quote marks, or words printed in *italic* font.

— *non-terminal symbols* : These are denoted by words typed in lowercase.

Each production rule begins with a non-terminal followed by an equal-sign and a sequence of meta characters, terminal, and non-terminal symbols, terminated by a period.
The meta characters imply:

Alternatives |	A vertical bar between symbols denotes the choice of either one symbol or another symbol. (i.e. a | b means either a or b)
Repetition {}	Curly brackets denote that the symbols in between may be present zero or more times. (i.e. { a } stands for: empty | a | aa | aaa | ...)
Optionality []	Square brackets denote that the symbols between them may present. (i.e. [a] stands for: a | empty)

Note also that all symbols can be tagged with an optional superscript. This superscript has no function, other then to distinguish otherwise identical symbols.

object-def	= *define tuple* "{" { comp } "}" identifier.
comp	= type-specifier identifier.
type-specifier	= *type* identifier | *set_of* type-specifier
	| *ref_to* type-specifier | *seq_of* type-specifier.

From this syntax specification, we see that a component of a tuple object can actually be (1) a primitive object, (2) a reference to a tuple object, (3) a set or sequence of tuple objects, or (4) any combination of the previous three possibilities. We will define a number of primitive (or build-in) types, e.g. strings, integers, floats, points, rectangles etc.

As an example of an object definition we will define a **node** as an aggregation of a name, a position and a set of references to ports. The name of the object is represented by a string. The position is represented by a point, i.e. an x and an y coordinate.

```
define tuple {
      type string n_name;
      type point  n_pos;
      set_of ref_to type port  n_ports;
   } node;
```

Similarly, a port could be defined as an aggregation of a name, a port-type, and a reference to a node. The port_type of a port contains e.g. information about the position of the port (for graphical representation) as well as about the type of the data value carried by the port.

```
define tuple {
      type string p_name;
      type port_type  p_type;
      ref_to type node  p_node;
   } port;
```

The metaschema in Figure C.2 can be extended to include the operations applicable to instances of a particular type. At this moment we assume that each type has a fixed set of operations, depending on whether it is a tuple, set, sequence or reference type (cf. section C.4). Note, however, that the implementation of these operations depends on the type.

Which primitive operations are available for a given type, depends on whether it is a *tuple*, a *set*, a *sequence* or a *reference* type. Objects that are tuples support the following operations †: *get*, *put*, *allocate*, *release*, and *delete*. The "get" and "put" operations hide the low level store and fetch functions (cf. section C.5.1). The operation "get" reads in an object from secondary storage, "put" puts it back, i.e. it updates the contents of the secondary memory to reflect the actual status of the object. "Allocate" and "delete" take care of the in-core storage management. Finally, "release" releases any objects referred to by the object, i.e. it puts them back in the database as soon as all references have ceased to exist. The semantics of these operations are explained in more detail in section C.4.1.

Set objects support the above operation as well. In addition a set object has operations to scan the elements of a set, to add and remove elements of a set, and to test for the presence of a particular element in the set. The syntax of the set operations is similar to the corresponding EC syntax.

Sequence objects are mostly similar to sets. The differences lie in the implementation. The elements of a sequence are stored in a separate file, and can't be referenced, i.e. they don't have a database key. Instead, they can be distinguished by their position in the sequence. Furthermore, we do not allow sequences of objects that contain references to other objects.

The primitive operations associated with a reference type are the same as those of a tuple, i.e. we can *get*, *put*, *allocate*, *release*, and *delete* an object via a reference. In addition however, a reference can also be **dereferenced**. The implementation of the dereferencing operator will take care of an important part of the data-management. When a reference gets dereferenced, the object pointed to by the reference is automatically fetched from the database, if not already available. The applications programmer will

† Note that this is only a minimal set of operations. The minimal set of operations is the set required to implement the data-management correctly.

generally be unaware of this.

The information in the metaschema is available to application programs. For example, an interactive user-interface could use the information to browse through the objects in the object pool.

C.2.1.2 The EC layer

The EC layer implements the translation from the data-type definitions at the user layer to compiled code. The reason we choose EC as a language to implement our data-type definitions is primarily the fact that EC allows the definition of parameterized data-types. This ability considerably simplifies the definition of types representing references to objects. The type inheritance mechanism incorporated in other languages, e.g. C++ [Stro85] or Objective C [Cox86], would require an additional preprocessor for generating these reference types.

Given a tuple definitions as shown above, we can easily define a cluster that implements the object type. In addition we can generate all necessary derived types, i.e. references, sets and sequences by simply instantiating the generic reference, set and sequence types with the appropriate parameters.

In order to show how this is accomplished we have to explain briefly the mechanism by which the EC programmer can define new types.

C.2.1.2.1 EC, sets and clusters

The features that make EC the desired choice for implementing the DMP package are chiefly:

1. EC supports sets and other high-level data structures.
2. EC is an extensible language. It is possible to define new data-types and to overload existing operators, as well as to redefine the set-oriented operators (exists .. in .., forall .. in .. suchthat .., etc)
3. EC operations can be specified either as macros or as procedure calls.
4. EC data-types can be conveniently parameterized. For example, it allows types as parameters.

By introducing sets and sequences it is possible to model the data handled by a design tool in a very universal way. By postponing the binding of data-types, including sets, to a representation, or by modifying it later, it is possible to make an optimal trade-off between flexibility and generality [Katz83a], without requiring more than a simple recompilation.

A new type is defined with a map statement which binds a cluster and a set of actual values for the cluster parameters, to the new type. An example of a map statement is:

map **cluster_name** (**cluster_arg_list**) *type* **new_types_name**;

A cluster definition itself consists of four parts:

1. *representation* : definition of the data structure used to represent instances of the types defined by the cluster.

2. *operations* : operations that can be applied to instances of the types defined by the cluster. Operations can be implemented either as procedures, or as macros. In the first case, the definition of the operation is preceded by the keyword **proc**, in the latter it is preceded by the keyword **oper**. Clusters can also redefine build-in operators, e.g. "+", "/", etc. To redefine a build-in operator, the name of the operator has to be preceded by the keyword **oper** as well.

3. *components* : A cluster can have one or more components. The components of an object that is an instance of a type defined by such a cluster, can be accessed similarly to the members of a C structure.

4. *constants* : definition of constants, e.g. **null** and **nil**. The constant **null** usually denotes the null instance of a type, whereas **nil** denotes an undefined pointer to an instance of the type.

The key to the simplicity of the DMP lies in the implementation of the sequence, set and reference types. Using the facilities provided by EC, it will turn out that three, appropriately parameterized, cluster definitions are sufficient to implement them. A new set, sequence or reference type is created by simply mapping the generic cluster to the new type, using the

type of the object as a parameter. As a result, all an application programmer has to do, is to specify the object types needed by a particular application program. For an example, the reader is referred to section C.3.

C.2.1.3 The database layer

The database layer takes care of the actual transfer of objects between secondary storage and the address space of a particular design tool. As said before, each object is characterized by a unique key, by which it can be fetched from the database. In addition, the database procedures allow objects to be clustered. This can be used to advantage when it is known beforehand that a particular collection of objects is related, so that when we want to fetch one of them, there is a high probability that we want to fetch the others as well. In particular, we use this feature to store objects that are in the same set of objects. The database procedures also maintain the consistency of the database in the sense that there can be no lost objects. All objects must be reachable starting from a baseset.

The procedures that implement the low-level database library, as well as the structure of a database are described in detail in section C.5.

C.3 Object Definition and Manipulation - An Example

In this section we will see how objects and object operations, including the data-management operations, are used in an application program. The example that we will discuss is taken from the design of a flowgraph editor. Based on this example we will see how the actual data-management operations are hidden from the application programmer, and how in fact the data-management has become transparent.

The procedure *hf_delnode()* deletes a node from a flowgraph. A flowgraph is an instance of a tuple type, that is defined as follows:

```
define tuple (
        type string g_name;
        set_of ref_to type node  g_nodes;
        set_of ref_to type port  g_ports;
        set_of ref_to type edge  g_edges;
    ) graph;
```

The node and port types are as defined in Section 2.1.1. The definition of the edge is:

```
define tuple {
        int delay_count;
        ref_to type port  i_port;
        ref_to type port  o_port;
    } edge;
```

The data-schema depicting the relations between the design objects is shown in Figure C.3.

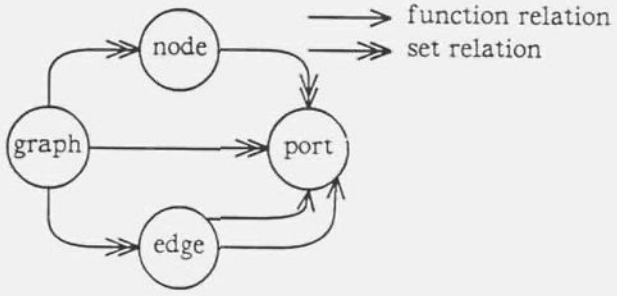

Figure C.3. Data-schema corresponding to the example

The procedure hf_delnode() (cf. Figure C.4) is rather simple. First it determines which node must be deleted. The operation "nearestnode" determines the node that is *nearest* to the given position. Next, i.e. if there is a node within a certain distance of the given position, the procedure determines for all ports of the node, whether there is an edge incident to it. If so, the edge is deleted. In addition, the other port connected to the edge is added to the set of ports of the graph referenced by G. If not, we can simply remove the port from the ports of the flowgraph. Here we see the power of EC's set operations. Finally the node itself is deleted.

We have given this code here to show that the the data-management is transparent to the application program, *and the application programmer.*

The application programmer doesn't have to know about the data-management procedures at all. The only thing he has to do, is to take care to release or delete the object references used in the procedures in accordance with his intentions.

```
# include <graph.h>

/*    This routine deletes a node from a graph
*/
operd void nearestnode(type graph, type nodep, type point);

void hf_delnode(G, screenpos)
   type graphp G;
   type point screenpos;
{
   type nodep   N;
   type edgep   E;
   type portp   P;

   nearestnode( G,N,screenpos);
   if(N != (type nodep)null) {
      forall P in ( N).n_ports do {
         if(exists E in ( G).g_edges suchthat ( E).i_port == P) {
            remove E from ( G).g_edges;
            add ( E).o_port to ( G).g_ports;
            delete(E);
         }
         else if(exists E in ( G).g_edges suchthat ( E).o_port == P) {
            remove E from ( G).g_edges;
            add ( E).i_port to ( G).g_ports;
            delete(E);
         }
         else {
            remove P from ( G).g_ports;
         }
      }
      remove N from ( G).g_nodes;
      delete(N);
   }
}
```

Figure C.4. Procedure hf_delnode()

In order to see how the data management is done, we have to look at the implementation of the reference types. Specifically we have to look at the operations that dereference, release and delete a reference. The procedure

hf_delnode(), uses four reference variables, resp. G, N, E and P. The trailing p's in the declarations of these variables, hint at the fact that an object of type ...p is a reference to a ... †. The procedure hf_delnode() uses these variables to access the components of the objects they point to. The component objects are found by dereferencing the references. Dereferencing is done using the (cluster defined) operator ~, which is (syntactically) similar to the unary * operator in C. The operation that implements the dereferencing is shown in Figure C.5. It is defined as part of the cluster used to define reference types. Except for the keywords, it resembles an ordinary C procedure definition. The type returned by the oper is specified; in this case the type is T‡, i.e. the dereferencing operator returns the object its first argument refers to. "_ref" denotes the name of the cluster containing the specification. Inside the cluster definition this name is used as the name of a type. A variable of type _ref is represented by a structure that has three members, resp. a pointer, p, of type T *, an object key, dbkey, and a pointer, dbacces.

```
oper type T oper ~ ( r )
    type _refs r;
{
    if( r.p == (type T) nil)
        if( r.dbkey == (type key) null)
           fatal error: dereferencing a null reference;
        else
           r.p = get ( r.dbacces, type T, r.dbkey);
    result *r.p;
}
```

Figure C.5. Dereferencing an object reference.

---

† In a similar way, a trailing s in a type name indicates a set of objects. This is an arbitrary convention. The only thing we need is some systematic way of deriving type names for sets, sequences and references of specified types.

‡ We will use the letter T as a type name. Note that in an EC cluster definition we can have type names as parameters. In fact, the code shown in Figure C.5 is taken directly from the corresponding cluster definition.

By looking at Figure C.5, we see that when we dereference an object reference, we first check whether the reference pointer is already defined, i.e. whether the pointer already points to the object. If not, we need to know the key of the object so that we can get it from the database as yet. The get operation checks to see if the object is already read in; if not it actually reads the object from the database and returns a pointer to it. If the object was already read in, get() just returns a pointer to the object. The implementation of get() depends on the type of the object to be read in. The type name T is given as the second parameter of the get operation.
The operation ~ returns the object pointed to by the reference. When an application programmer attempts to dereference an undefined pointer, i.e. a reference to an object whose key is not defined, an appropriate error-message will be generated.

The consistency of the data-management procedures is based on the fact that all objects keep track of the number of references pointing to them. Hence we require that the application programmer accesses an object only via a corresponding object reference. Then, by a proper definition of the assignment operation of the reference type, it is easy to keep track of the total number of references to an object. The reference count is kept as the sum of the active, passive and database reference count of the object. A passive reference is a reference by key only, i.e. a reference to an object that has not yet been fetched from secondary storage. An active reference is a reference whose pointer has been set to point to the object. The database reference count keeps track of the number of references to the object in the database.

The procedures release and delete, hide the other part of the data management, i.e. they decide if and when an object must be written back to the database. This decision is based on the values of the reference counts of an object. For example, when an object is no longer referenced, i.e. the sum of all reference counts is zero, it can be deleted. An outline of the procedure release is shown in Figure C.6. Since the procedures release and delete are almost similar we show only the procedure release.

```
oper void release ( r )
   type _ref r;
{
   if(r.p != (type T) nil) {
      MARK(r);
      if((r.p->a_rcount == r.p->mark_count)) {
         /* this is the last active reference */
         UNMARK(r);
         r.p->a_rcount--;
         r.p->db_rcount++;
         if(OBJISTOUCHED(r.p)) {
            if((r.dbkey = r.p->dbkey) == dbNILKEY) {
               r.p->dbkey = dbNewKey(r.dbacces,USEROBJ);
               r.dbkey = r.p->dbkey;
            }
            put(r.dbacces, r.p);
            release(r.p);
         }
      }
      else {
         UNMARK(r);
         RESETSTATUS(r.p,MARK_CHECKED);
         r.p->a_rcount--;
         r.p->db_rcount++;
      }
   }
   r.dbkey = dbNILKEY;
   r.p     = (type T) nil;
}
```

Figure C.6. Release of an object reference.

The procedure "release" decrements the active reference count of an object, and increments its database reference count. By checking the number of active references to the object (a_rcount) the release operation determines whether the object must be put back in the database. From Figure C.6 it follows that an object is put back in the database when the active reference count is equal to the so-called mark count, which is set by the procedure MARK(). The mark_count has to be introduced to determine how many active references remain after the object is put back in the database. This is done by tracing all objects that can be reached from the object, and incrementing their mark_count. If the mark_count of an object is equal to

its active reference count, this means that we have the last active reference to the object. The actual determination can be complicated, because the object, or any of the objects referenced by it may refer back to the object to be put in the database.

The procedure "release" is used to let the data management routines know that the reference is not needed anymore. The object it refers to can be put back in the database whenever that is convenient. The procedure "delete" is almost similar to "release", except that the db_rcount of the object is not incremented. If it becomes zero, the object is deleted, i.e. the storage allocated for the object is freed, and the object is removed from the database.

C.4 Implementation of Tuple, Set, Sequence and Reference Types

In the previous section we saw how the data-management is handled automatically via the operations that dereference, respectively release or delete a reference to an object. In this section we will discuss the internal representation, i.e. the actual implementation, of objects, sets of references to objects, sequences of objects, and references to objects.

Refering to Figure C.1 we are now looking at the ellipse named *Data-type map*. Starting point for this are the tuple definitions given in the previous section, from which we have to derive all type declarations needed to implement the tuple objects†. This includes the type definitions for the tuples, and all types needed to implement sets, sequences and references of all tuples, the so-called derived types.

For every type, we will generate a header file that specifies the implementation of the type, as well as its so-called *external representation*. The external representation specifies the properties of the variables in an

† Note that all code shown in this section is also actual EC [Katz85] code, that is compiled by the EC compiler.

application program. For example, to specify the external representation of a (node) reference we can use a C typedef statement as shown below:

```
typedef type node * nodep;
```

The statement above gives a name to the reference type. This is done systematically, we add a 'p' to the original typename to represent a reference, a 's', to represent a set, and a 'S', to represent a sequence. The external representation of a tuple is almost similar to that of its original definition. The only change is a change of name for derived types, as explained above.

```
typedef tuple (
   type string n_name;
   type point  n_pos;
   type ports n_ports;
) node;
```

The type of the component n_ports has become 'ports', to represent the fact that it is a set of references to ports. The type ports is declared in a separate map statement, as explained next.

In order to define a new type in EC, the programmer has to specify :

1. the external representation, e.g. in the form of a tuple, oneof, set or sequence definition, and

2. the implementation.

The implementation of a type is specified by a map statement, which is an EC statement that is used to bind a type to an instance of a specific cluster. The power of the map mechanism lies in the fact that cluster parameters can be type names. For example, to specify the implementation of a (node) reference we use an EC map statement as shown below:

```
map _ref ( type node ) type nodep;
```

This expresses the fact that nodep is an instance of _ref with parameter "node". It is important to note that all other references can be implemented the same way. Similarly, to implement a reference to a port we will have:

```
map _ref ( type port ) type portp;
```

Therefore, a single cluster '_ref' is sufficient to implement all possible reference types. The same is true for all other derived types as well: It will be clear that the possibility of defining clusters simplifies the implementation of the DMP.

The implementation of a tuple is slightly more complicated. Here we have to define the cluster as well. However, all clusters implementing tuples have the same structure. The only differences between tuples are in the number, type and names of the components. As a result, it is not difficult to write a program that does this automatically. The program will consult a database, containing all relevant data about previously defined types. In addition we may want to define specialized operations for the types implemented by these clusters. This can be implemented as part of the program generating the cluster definition required to implement the tuple type. This program can simply prompt the application programmer to specify additional operations. The program can also take care to include special preprocessor directives, that allow separate compilation of the cluster.

As said before, the implementation of a type is specified by binding it to a cluster definition. In what follows, we will show the structure of a header file, that contains a complete specification of a node tuple. All code shown in the remainder of this section is part of a header file called 'node.h', which is located in a special include directory, so that all application programs can include it.

```
/* ... start of file defining node tuple */
/* header files to define types of components */
# include <string.h>   /* define type: string (strings of characters) */
# include <point.h>    /* define type: point (x,y point) */
# include <ports.h>    /* define type: set of references to port */

/* header files to include the definitions of the generic clusters */
# include <set.h>
# include <ref.h>

/* header file to include DMP interface library definitions */
# include <dmp.h>

# define NodeRadius   20   /* addtl. definition */
```

The external type definition of the node is given next. A node is a tuple

with three components, resp. a name, a position, and a set of ports. The difference between a tuple and an ordinary C struct declaration is that a tuple does not imply a particular storage structure. In fact the external representation can have components that are not part of the internal representation associated with the object. In such a case the value of the component has to be computed anew every time it is accessed.

```
typedef tuple {
   type string  n_name;      /* the name of the node */
   type point   n_pos;       /* the (x, y) position of the node */
   type ports   n_ports;     /* the ports of the node */
} node;
```

Next we get the declaration of the cluster. Clusters implementing tuple objects do not have parameters.

```
cluster _node ( )
{
   rep type _rnode n;
```

The representation of a node, i.e. the data-structure used to implement node objects is defined above. The definition above specifies that a node is represented by an object of type _rnode. This type is defined below, using an ordinary C typedef statement. Note that this type is defined in the cluster body. It is local to the cluster, and will not be visible outside it.

```
   /* ... local type definitions ... */
   typedef struct {          /* in-core representation of a node */
      type string n_name;    /* rep of name of node */
      type point  n_pos;     /* rep of (x, y) position of node */
      type ports  n_ports;   /* rep of ports of node */
      short   a_rcount;      /* # of active pointers to object */
      short   p_rcount;      /* # of passive pointers to object */
      short   db_rcount;     /* # of pointers to object in db */
      short   mark_count;    /* mark count of object */
      short   status;        /* status of object */
      type key  dbkey;       /* database key of object */
   }  _rnode;
```

The type definition above, has more members than the node tuple that is mapped to it. These extra members hold the information needed by the data-management procedures. In fact, there is no predefined relation

between the members of the internal and external representations. The only constraint is that the internal representation has the additional members named a_rcount, p_rcount, db_rcount mark_count, status, dbkey and dbacces. They are used to keep track of the number of references to the object, as well as the database in which the object is stored. What is important is that the cluster provides the operations to compute each of the components of the external representation, i.e. a cluster used to implement a node must have components named n_name, n_pos and n_ports.

Next we can have a number of additional data structure definitions, which are used by the cluster operations. In this case we only need a structure to define the database representation of a node.

```
typedef struct {   /* database object associated with a node */
   type key   n_name;  /* db rep of name of node */
   type point n_pos;   /* db rep of (x, y) position of node */
   type key   n_ports; /* db rep of ports of node */
} _dbnode;
```

The database representation is used by the get and put operations to store and fetch the object in resp. from a database. It differs from the external representation, in the sense that all object references, as well as strings are replaced by database keys (dbkeys).

The definition of the object operations, shown below, is fairly straightforward. They are defined as part of the cluster body, following the keywords **oper** or **proc**. A "proc" operation will be implemented as a function call, whereas an "oper" operation will be implemented as a macro. The body of an oper definition actually replaces its call in an application program. Instead of a return statement, it has a result statement with the same function. Except for the change in name, the function is the same however. Build-in operators can be redefined. The difference with an ordinary definition is that the name of the operation, the operator to be redefined, is preceded by the keyword oper.

The operations "allocate" and "delete" handle the in-core storage management of an object. They take care of proper initialization, resp. cleaning up when an object is allocated, resp. deleted. **Cmode** is an EC keyword, whose type

is that of the cluster. It is used to distinguish between operations, in different clusters, that otherwise could not be distinguished from one another, i.e., they have the same name and the same arguments. SETSTATUS is a special macro that sets the status of an object, according to its second argument. In the case shown below, the status of the object is made DELETED. This is done to prevent an object from being deleted twice, e.g. if the object contains a reference to an object that in turn contains a reference to the first object.

```
/* ... Operations ... */
oper type _node * oper allocate ( cluster_name )
   cmode cluster_name;
{
   type _node * res;
   res = (type _node *)malloc(sizeof(type _node));
   res->n_name &= ""; res->n_ports = (type ports) null;
   result res;
}


oper void oper delete ( p )
   type _node * p;
{
   SETSTATUS(p,OBJDELETED);
   delete ( p->n_name ); delete ( p->n_ports );
}
```

The delete operation does not actually free the storage allocated to the object. This is left to the operation that deletes the last reference to the object. The reason for this is that we don't want the cluster defining the implementation of the tuple to contain details of the storage management. The delete operation defined in the _ref cluster (cf. section C.4.2) will also check the active and passive reference count of the object, and free the storage only if these are both zero. If not, an error message will be generated.

```
proc   type _node * get ( dbacces, cluster_name, dbkey )
   type DbAcces * dbacces; cmode cluster_name; type key dbkey;
{
   type _node * res;    /* result object (node) pointer */
   type _dbnode * dbo;    /* database object pointer */

   res = (type _node *)malloc(sizeof(type _node));
   /* fetch database object of node */
   res->db_rcount = dbFetch(dbacces,dbkey,&dbo);
   res->n_name &= get(dbacces, type string, dbo->n_name);
   res->n_ports = get (dbacces, type ports, dbo->n_ports);
   res->n_pos = dbo->n_pos;
   res->dbkey   = dbkey;
   res->mark_count = 0;
   INITSTATUS(res,OBJ_LOCKED); cfree(dbo);
   return res;
}
```

The operation "get" handles the fetching of an object from the database. The object is identified by its dbkey. "get" keeps track of the objects fetched from the database via the functions dbKeyItsPtr() and dbAddKey(). These routines, together with the function dbRemKey(), maintain a hash table that, for every database used by the applications program, relates the database keys of the objects to their in-core pointers. If an object is not yet fetched from the database, its pointer will have the value nil. In that case, the routine dbAddKey and dbRemKey will keep track of the number of nil pointers to the object. This is necessary in order to make the reference count of the object consistent once it is read in from the database. The number of nil pointers is returned by the function dbAddKey(). The functions dbFetch(), dbFetchD(), dbCreateD() and dbStore() are defined in section C.5.1. They are part of the DMP interface library. In this case, dbFetch will access the database, and read in the object denoted by the object key. It will allocate storage for the object, and return a pointer to this address as the value of 'dbo'.

Note that for different object types, the implementation of the get operation will differ only in the type and number of components that need to be fetched.

```
proc type key put ( dbacces, p )
   type DbAcces * dbacces; type _node * p;
{
   type _dbnode dbrep;

   RESETSTATUS(p,OBJ_TOUCHED);
   dbrep.n_ports   = put(dbacces, p->n_ports);
   dbrep.n_name    = put(dbacces, p->n_name);
   dbrep.n_pos     = p->n_pos;
   dbStore(dbacces,p->dbkey,&dbrep,p->a_rcount + p->p_rcount +
            p->db_rcount,sizeof(type _dbnode));
   RESETSTATUS(p,OBJ_LOCKED|OBJ_NEW);
   return p->dbkey;
}
```

The operation "put" recursively puts all components of the object back in the database, and then stores an object relating the components of the object to the object itself, via their respective database keys. The data-structure stored in the database is the database representation of the object (see local type declarations above).

The programmer who defines the cluster is free to add additional primitive operations. An example is given below. The operation nearestport, computes the port in the set of node ports that is nearest to a given point.

```
proc type portp nearestport ( n, pnt )
   type _node n;
   type point pnt;
{
   type portp p1, p2;
   int dmin, tdmin;

   dmin = 4 * NodeRadius;
   p2 = (type portp)null;
   forall p1 in n.n_ports suchthat
         ((tdmin = dist((~p1).p_pos,pnt)) < dmin) do {
      dmin = tdmin;
      if(p2 != (type portp) null)
         delete(p2);
      p2 = p1;
   }
   return p2;
}
```

The following part of the cluster body consists of the definitions of the components of the tuple type. For every component in the tuple definition we have to specify a 'comp', denoted by preceding it with the keyword **comp.**. Syntactically, components are distinguished from oper and proc definitions only by the initial keyword. Here we show only the comp definition of the first node component.

```
comp type string n_name ( p )
   type _node p;
{
   (p.n_name)
}
```

The last part of the definition of a tuple type is the definition of constants. For tuple types we will define the constant **nil**.

```
    constant type _node * oper nil (cluster_name)
        cmode cluster_name;
    {
        ((type _node *)0)
    }
}
```

This finishes the definition of the cluster. There is only one thing left to do and that is to map the cluster to the node type. This is done using a map statement as shown below.

```
    map _node    ( )    type node;
/* ... end of file defining node tuple */
```

C.5 Building a Database based on the DMP Data Abstractions

In this section we will discuss the interface functions needed to build a database using the methodology to define and implement types defined in the previous sections. The high-level interface will implement the metaschema shown in Figure C.2. To do so, we will have to define a number of clusters. These will be shown, in outline, in section C.5.2. We start with a discussion of the DMP interface library. This library contains the low-level access functions needed to implement the "get" and "put" operations of the tuple types.

C.5.1 The DMP Interface Library

The DMP package contains a low-level library, with routines that can create, delete and modify objects. Depending on the efficiency desired, objects can further be subdivided into e.g. descriptor, set, sequence and tuple objects. At this moment we distinguish only descriptor objects. All other objects are called user-defined objects. User-defined objects are arrays of bytes with no predefined interpretation. Descriptor objects are used to store strings.

The file structure of a DMP database is shown in Figure C.7.

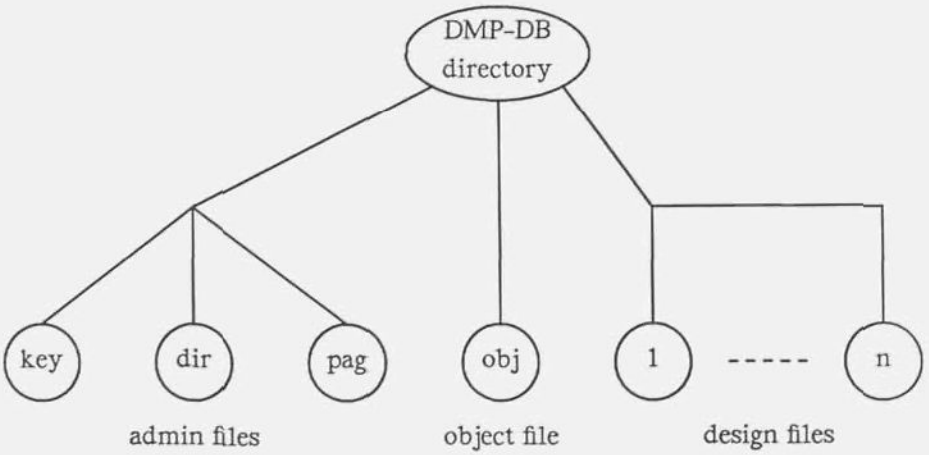

Figure C.7. File Structure of a DMP Database.

All objects are identified by a unique object key. The usage of object keys is administrated in a bitmap that is stored in the "key" administrative file . A new object key is allocated by calling the routine dbNewKey(). The DMP interface routines use two additional files, the "dir" and "pag" files, to store a hash table that contains basic information about the object, e.g. its key, the address on which the object can be found, etc. Finally there is a fourth file that contains the objects. All transaction with these files are atomic. This is done by using semaphore operations. The design files are intended for storing sequence elements. With every sequence object there is a corresponding design file, the records in these files correspond to the elements of the sequence.

The operations implemented sofar, with a specification of their function, are discussed below. The types of the values returned by the functions, and the types of the arguments are indicated. The return type is the type preceding the ← arrow. All routines take as their first argument a value of type DbAcces, which is a pointer to a structure containing all information

necessary to access a particular database; it also contains the buffers required to make the manipulation of the database files efficient. For clarity, these details are left out of the argument lists. The type DbKey stands for a database key, type DbKeySet stands for a set of database keys. Type DbMode is an enumerated type. It has values *PERMANENT_LOCK* and *ATOMIC_LOCK*, so as to allow a database to be locked either for a single transaction, or for a sequence of transactions.

The functions in the DMP interface library are:

1. To allocate a new key for an object:
   type DbKey ← dbNewKey()
   The function dbNewKey() allocates a unique key for an object.

2. To create objects:
   *type DbKey ← dbStore(type DbKey objkey; char * object; int rcount, lmode, size)*
   The function dbStore() is used to create and replace user-defined objects. If the object denoted by argument objkey exists it is replaced by the new object, pointed to by object. If the object denoted by objkey does not exist, it is created. The *rcount* argument is used to pass the number of references to the object to the data-management routines. Next the *lmode* is used to specify the whether the object should remain locked. If so, the function of the store operation is just to update the contents of the database files. The final argument is the *size*, i.e. the number of bytes, of the object.
   type DbKey ← dbStoreD(type string descriptor)
   The function dbStoreD() is used to store and/or create descriptor objects. It creates a descriptor object for the symbolic identifier denoted by the string *descriptor*. The function returns the key of the newly created object.

3. To fetch objects:
   *(int) ← dbFetch(type Dbkey objkey, char ** object; int lmode)*
   The function dbFetch() is used to fetch the object denoted by the *objkey* from the database. If the object is not to be locked, the access is

read-only. The database reference count of the object is returned. A pointer to a pointer to the object is implicitly returned in the *object* argument.

*(char *) ← dbFetchD(type Dbkey objkey)*

The function dbFetchD() returns the descriptor associated with *objkey* as a character string.

4. To unlock objects:

   (void) ← dbUnLock(type DbKey objkey)

   The function dbUnLock() is used to unlock an object that was locked in the database during a dbFetch(). The object is identified by its *objkey*.

5. To delete objects:

   (void) ← dbDelete(type DbKey objkey)

   The function dbDelete() is used to delete the object denoted by the *objkey*. Depending on the class the object belongs to, dbDelete() performs all actions necessary to delete the object.

6. To access a database:

   — To open it:

   type DbAcces ← dbOpen(type string dbname; type DbMode amode)

   The function dbOpen() is called to create a new database descriptor. It allocates and returns a DbAcces structure to be used by subsequent operations that access the contents of the database. The *amode* argument selects between permanent and atomic locking. Atomic locking implies that every operation needs to lock the database. On the other hand, if the database is permanently locked, it is locked once when the database is opened. This lock is then removed when the database is closed. The locking mechanism guarantees that at any one time only a single process can have access to the database. Note that the function dbOpen does not need the DbAcces argument required by all other ODM functions.

— To close it:

(void) ← dbClose(type DbAcces dbacces)

The function dbClose closes the database, i.e. it removes any permanent lock, flushes the buffers, and closes the files containing the database objects.

A major extension of the DMP library will be the addition of functions to store and fetch sets and elements of sets. The main purpose of this is to increase the efficiency by taking advantage of the fact that once we access a set, we will certainly want to access the elements of the set as well. The efficiency of accessing the set-elements can be increased if, together with set, we store information about where the set-elements are located. In fact we have to duplicate the information normally contained in the hash-tables stored in the pag and dir files. In addition we can take care to store the set-elements together, so that the number of page accesses becomes minimal.

C.5.2 Implementing the DMP Metaschema

Looking at Figure C.2 we see that a DMP database consists of a collection of type definitions. The type definitions are instances of a tuple, **dbtype**. In addition to defining a cluster to implement this tuple, we define a cluster that implements operations to open, resp. close a database. This cluster will have the set of type definitions as one of its components. That way, it becomes possible to apply the usual set operations, e.g. *exists*, to select a particular type, thereby gaining access to the baseset of the type.

A typical piece of code to open a database, access the baseset (see Figure C.2) of one of the dbtypes, and close it again is shown in Figure C.8.
Except for the already mentioned open and close operations, we see that we have used an operation that returns the baseset of a type. This operation is added to the set cluster. In addition to returning the baseset this operation will add some extra information to the tuple describing the type. The extra information, the addresses of some access functions, is used to write the baseset back to the database, once it is closed. To that end we also have to modify the put and release operations of the dbtype tuple.

Finally, the definition of the dbtype tuple:

```
{
   type dbtypedb dmp_db;
   type graphs   graph_baseset;

   dmp_db = open(type dbtypedb, "pathname", lock_mode);
   graph_baseset = baseset(dmp_db.types, type graph);
         |
         |
      /* Additional code */
         |

   close(dmp_db);
}
```

Figure C.8. Accessing a DMP database

```
define tuple {
   type string t_name;
   type key    t_baseset;
   set_of ref_to type comp  t_comps;
} dbtype;


define tuple {
   type string c_name;
   ref_to type dbtype c_type;
} comp;
```

Given these definitions, and looking at the code of Figure C.8, we see that it is quite easy to access and manipulate a database. The application programmer simply opens it, selects the basesets to be used, and does whatever is required with the objects referenced by the basesets. Finally, the programmer closes the database, which causes all basesets to be written back, and the contents of the files to be updated. In between the programmer can control the amount of information that is kept in-core by carefully applying put and/or release operations. At no time however, the programmer is required to take care of the details of the data management. These details are hidden in the operations that allocate, release, put and/or delete objects, object references etc.

C.6 Discussion

The data-management approach shown in this paper is based on the application of modern software engineering techniques for structuring large programs. By modeling (VLSI) design data as a collection of typed objects, and by structuring this collection using sets and interobject references it is possible to create a database and to access this database via access functions derived from the definition of the object types. The implementation discussed here was done with *Enhanced C*, a high-level set-oriented extension of the C programming language. The data structuring methods offered by EC are well matched to the requirements posed by the data management method. In particular, the possibility of defining parameterized data-types and the overloading of operators are drawn upon. EC makes it possible to define new data-types by defining clusters. Clusters are mapped to a particular data-type, by providing a set of actual values for the cluster parameters. The flexibility of EC derives in part from the fact that these values can be almost anything, from simple integers and reals to type names and even entire statements.

New object types are easily defined. In addition, types implementing references to objects and sets of references to objects are derived automatically from the initial object type definition. As a result, the amount of programming to be done to define a new object type and to integrate it with the rest of the system is minimal.

The implementation of any particular set of objects can be matched to the application at hand by defining alternative clusters for implementing sets. As long as the external specification of the object types are not changed, no reprogramming is necessary. Simply recompiling the affected software is sufficient.

The work described in this paper is now being extended in several directions. First of all we are working on an advanced set of routines for doing the low-level storage, as described in section C.5. The new routines incorporate sophisticated buffering schemes, and will be able to take advantage of the clustering of objects in sets. The second direction is to

```
{
   type dbtypedb dmp_db;
   type graphs   graph_baseset;

   dmp_db = open(type dbtypedb, "pathname", lock_mode);
   graph_baseset = baseset(dmp_db.types,type graph);
         |
         |
      /* Additional code */
         |

   close(dmp_db);
}
```

Figure C.8. Accessing a DMP database

```
define tuple {
   type string t_name;
   type key    t_baseset;
   set_of ref_to type comp  t_comps;
} dbtype;


define tuple {
   type string c_name;
   ref_to type dbtype c_type;
} comp;
```

Given these definitions, and looking at the code of Figure C.8, we see that it is quite easy to access and manipulate a database. The application programmer simply opens it, selects the basesets to be used, and does whatever is required with the objects referenced by the basesets. Finally, the programmer closes the database, which causes all basesets to be written back, and the contents of the files to be updated. In between the programmer can control the amount of information that is kept in-core by carefully applying put and/or release operations. At no time however, the programmer is required to take care of the details of the data management. These details are hidden in the operations that allocate, release, put and/or delete objects, object references etc.

C.6 Discussion

The data-management approach shown in this paper is based on the application of modern software engineering techniques for structuring large programs. By modeling (VLSI) design data as a collection of typed objects, and by structuring this collection using sets and interobject references it is possible to create a database and to access this database via access functions derived from the definition of the object types. The implementation discussed here was done with *Enhanced C*, a high-level set-oriented extension of the C programming language. The data structuring methods offered by EC are well matched to the requirements posed by the data management method. In particular, the possibility of defining parameterized data-types and the overloading of operators are drawn upon. EC makes it possible to define new data-types by defining clusters. Clusters are mapped to a particular data-type, by providing a set of actual values for the cluster parameters. The flexibility of EC derives in part from the fact that these values can be almost anything, from simple integers and reals to type names and even entire statements.

New object types are easily defined. In addition, types implementing references to objects and sets of references to objects are derived automatically from the initial object type definition. As a result, the amount of programming to be done to define a new object type and to integrate it with the rest of the system is minimal.

The implementation of any particular set of objects can be matched to the application at hand by defining alternative clusters for implementing sets. As long as the external specification of the object types are not changed, no reprogramming is necessary. Simply recompiling the affected software is sufficient.

The work described in this paper is now being extended in several directions. First of all we are working on an advanced set of routines for doing the low-level storage, as described in section C.5. The new routines incorporate sophisticated buffering schemes, and will be able to take advantage of the clustering of objects in sets. The second direction is to

build an interactive user interface written in LISP. This interface will be able to give the same functionality to the designer browsing through the database as the application programmer has available via the EC interface. In addition, parts of this interface will be used to automate the definition of clusters from the tuple definitions shown in sections C.2 and C.3. Finally we are looking into the possibility of interfacing to other language environments, e.g [Cox86]..

Finally, we believe that the work described in this paper shows that it is possible to make the data-management involved in the design and implementation of VLSI design tools transparent to the tool programmer.

Acknowledgement:
The authors would like to thank prof. dr. J. Katzenelson, Technion - Israel Institute of Technology, for making the EC compiler available to them, and for carefully reading an earlier draft of this paper.

References

Afsa84. Afsarmanesh, H. and McLeod, D., "A Framework for Semantic Database Models," *Proc. NTU Symposium on New Directions for Database Systems*, (May 1984).

Anne86. Annevelink, J., "A Hierarchical Design System for VLSI Implementation of Signal Processing Algorithms," pp. 371- 382 in *Computational and Combinatorial Methods in Systems Theory*, ed. C.I. Byrnes, A. Lindquist, North-Holland (1986).

Bato85. Batory, D.S. and Kim, Won, "Modeling Concepts for VLSI CAD Objects," *ACM Trans. on Database Systems* **10**(3) pp. 322-346 (September 1985).

Bekk83. Bekke, J.H. ter, *Database Ontwerp*, Stenfert Kroese b.v. (in Dutch) (1983).

Cox86. Cox, Brad J., *Object Oriented Programming: An Evolutionary Approach*, Addison Wesley (1986).

Date81. Date, C.J., *An Introduction to Database Systems*, Addison-Wesley Systems Programming Series (1981).

Katz83a. Katzenelson, J., "Higher Level Programming and Data Abstractions - A Case Study Using Enhanced C," *Software Practice and Experience* **13** pp. 577 - 595 (1983).

Katz83. Katzenelson, J., "Introduction to Enhanced C," *Software Practice and Experience* **13** pp. 551 - 576 (1983).

Katz85. Katzenelson, J., *The Enhanced C Programming Language Reference Manual*, Technion - Israel Institute of Technology, Haifa 32000, Israel (March 21, 1985).

Kern78. Kernighan, B. and Ritchie, D., *The C Programming Language*, Prentice Hall (1978).

McLe85. McLellan, P., "Effective Data Management for VLSI Design," *Proc. 22nd IEEE/ACM Design Automation Conference*, pp. 652-657 (July 1985).

McLe83. McLeod, D., Narayanaswamy, K., and Rao, K.V. Bapa, "An Approach to Information Management for CAD/VLSI Applications," *Proc. Databases for Engineering Applications - ACM Database Week*, (1983).

Stro85. Stroustrup, B., "The C++ Programming Language - Reference Manual," *AT&T Bell Laboratories*, (1985).

Wirt77. Wirth, N., "What can we do about the unnecessary diversity of notation for syntactic definitions ?," *Comm. ACM* 20(11) pp. 821 - 823 (Nov. 1977).

Curriculum Vitae

Jurgen Annevelink werd geboren op 2 november 1959 te Laren (Gld) thans Lochem. Na het behalen van het VWO diploma in 1977 aan de scholengemeenschap Prins Alexanderpolder te Rotterdam, studeerde hij elektrotechniek aan de Technische Universiteit Delft. In augustus 1983 behaalde hij met lof het diploma van elektrotechnisch ingenieur. Het afstudeerwerk was getiteld: "A Hierarchical Layout to Circuit Extractor using a Finite State Approach". Van 1 september 1983 tot 1 januari 1988 was hij werkzaam bij de vakgroep Netwerktheorie, Faculteit der Elektrotechniek van de Technische Universiteit Delft. Vanaf 11 januari 1988 is hij werkzaam bij Hewlett-Packard Laboratories, Palo Alto, Californie, in een groep welke bezig is met het ontwikkelen van een prototype object-oriented database systeem.