GRAPE: A CASE Tool for Digital Signal Parallel Processing

Rudy Lauwereins, Marc Engels, Jean Peperstraete
Eric Steegmans, Johan Van Ginderdeuren

After an introduction situating Computer Aided Software Engineering (CASE) in general and indicating the possiblities of CASE for digital signal processing (DSP), a design example clarifies the development stages of a typical DSP application. A large part of this article is devoted to an overview of existing development tools for DSP. Finally, the CASE tool GRAPE (GRAphical Programming Environment) is presented, which allows for easy programming, compiling, debugging and evaluating high frequency real-time DSP systems. Its main distinctive feature is that the tool spans the whole design process, ranging from analysis over simulation and emulation up to implementation on general purpose DSP multiprocessors or integration on an Application Specific Integrated Circuit (ASIC). The DSP multiprocessor can be the target hardware or can be used for real-time emulation or accelerated simulation of an ASIC.

COMPUTER AIDED SOFTWARE ENGINEERING (CASE) tools reduce the effort needed to specify, compile, debug, run and document software projects by integrating all the tools required in the complete life-cycle of the project [Ho..1]. They help system designers to express their ideas easily, rigorously and consistently. In addition, they allow them to manage the huge complexity inherent in software analysis, design, implementation and verification. The box "CASE in general" presents a strict definition of what the authors mean with CASE. It also offers a non-exhaustive list of tasks to be carried out by CASE tools.

This paper focuses on the use of CASE tools for stream-oriented real-time digital signal processing (DSP) applications like they are found in such fields as telecommunications, consumer electronics, instrumentation, etc. These applications are characterized by a continuous stream of data samples or a continuous stream of blocks of data samples arriving at the processing facility at time instances completely determined by the outside world. Throughput requirements, input-output interfacing and degree of inherent parallelism are quite distinct from traditional data processing. The data can be processed either to determine a time critical control action to be taken—in which case the processing delay must be minimized to avoid stability problems in the controlled system, or to heavily process the samples itself—in which case the global throughput must be maximized. An example where both types of processing are carried out on the same stream of samples can be found in compact disc players [Cara1]. The raw data stream, which is read by the laser beam, is decoded, checked and corrected against errors, and filtered before it is passed to the audio amplifiers. In addition, this same data stream is processed to control position and focus of the laser beam and speed of the disc.

CASE in general

Definition: A CASE environment is a **consistent** and **integrated** set of tools to manage a software project on all its levels of detail throughout its whole life-cycle. What makes CASE tools different from non-CASE approaches is their level of consistency and integration:

- the use of a single database to represent all aspects of the software project allows for automatic propagation of design data from one representation to the other: re-inputting the same data for use in another utility can hence be avoided, reducing the chance for inconsistency.
- extensive cross-checking of the information available in different representations can be provided, as well as checking the completeness of the specifications.

List of tasks:

- executable specification
- simulation
- code generation
- compilation
- verification
- test pattern generation
- source level debugging
- monitoring
- preparation of documentation
- logging changes to the design specifications
- synthesis of algorithms (application generation)
- optimization
- ...

0740-7467/90/0400-0032$1.00©1990 IEEE

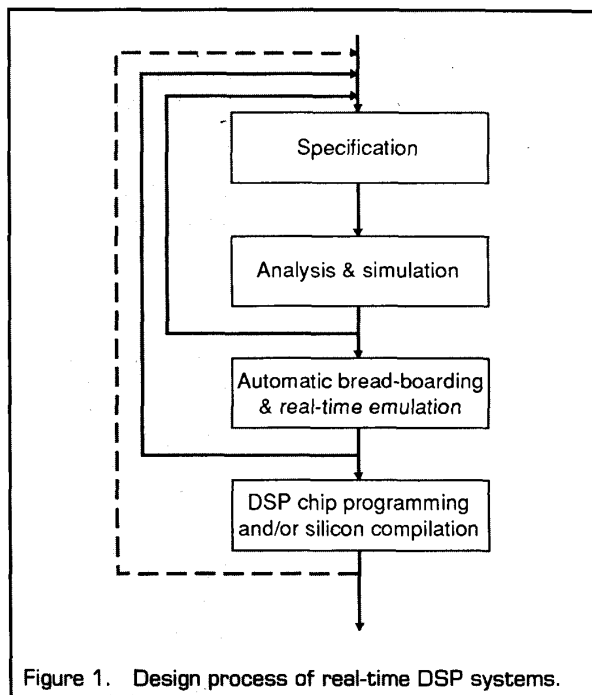


Figure 1. Design process of real-time DSP systems.

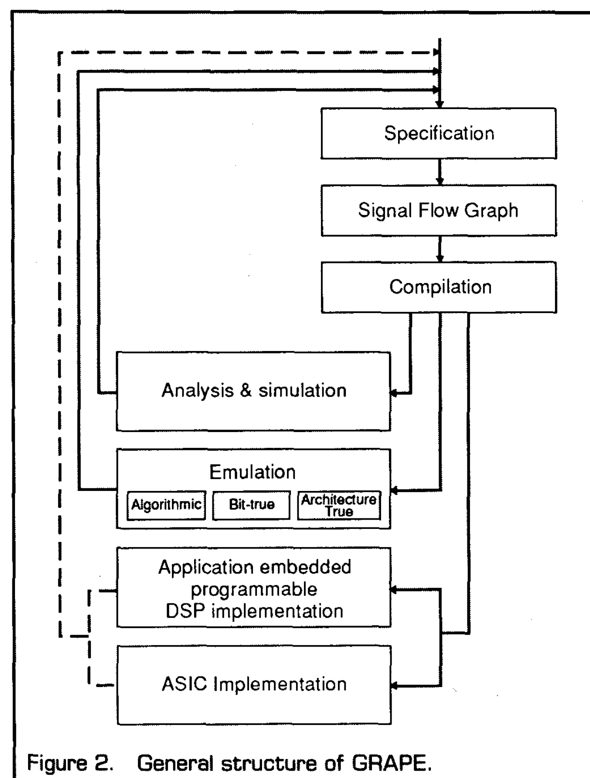


Figure 2. General structure of GRAPE.

The ever shortening product life cycles of such DSP applications put an increasing demand on the research and development (R&D) tools. Therefore, GRAPE has been developed as a CASE tool for the design of stream-oriented real-time digital signal processing algorithms and for their implementation on a DSP multiprocessor or on ASICs. This design process consists of four major steps, as depicted in Figure 1. After the specification of the algorithm, theoretical analysis and simulation give feedback to the system designers and allow them to modify the algorithm. Fine tuning can be done during the real-time emulation phase. Finally, the algorithm is implemented on a general purpose DSP multiprocessor or compiled to silicon.

To accomplish this iterative design process, the CASE tool GRAPE integrates utilities for all four steps. This leads to Figure 2 for the general structure of GRAPE.

In a first design step of a typical DSP project, the specification of the application algorithm—internally in GRAPE represented as a signal flow graph—is analyzed to predict its performance characteristics. The signal flow graph may be entered by the designer or generated from frequency domain specifications. Next, the same specification is simulated slower than real-time. Also, the surrounding equipment that will generate the sample stream to the hardware implementing the algorithm, and which will consume the processed data stream, is simulated. In the next step the application is traditionally bread-boarded in real-time. A much more flexible way to achieve this is to employ a reusable set of general purpose DSP processors. In general, the real-time requirements will make the use of a multiprocessor inevitable.

GRAPE thus includes a set of tools to partition an application in tasks and to assign these tasks to the available processors. When communicating tasks are assigned to different processors, GRAPE automatically includes communication primitives. Simulation of the surrounding equipment is gradually replaced by hardware as prototypes become available. This makes it possible to develop and test the environment of the ASIC before the first silicon becomes available. Finally, the single specification can be implemented on a general purpose (multi-)DSP, embedded in a product, or is compiled to silicon. The description of the simulator and the silicon compiler goes beyond the scope of this article on CASE. More details however can be found in [Sche1], respectively [DeMa1].

The emulation phase can further be divided in three sub-phases: algorithmic, bit-true and architecture-true emulation. In *algorithmic* emulation, correctness of the specification of the algorithm is verified, using an abundant word length for all computations, restricted naturally to the word length of the emulation engine. In a next step, the influence of restricted word length is analyzed in the bit-true emulation phase. This is very important in order to select target signal processors with minimal word length or to reduce silicon area on an ASIC. Finally, the hardware implementation of the algorithm could be checked in the architecture-true emulation phase.

Currently, GRAPE contains tools for algorithmic and bit-true emulation on a parallel computer. The software tools and the programmable hardware for architecture-true emulation are in development. This hardware will consist of a printed circuit board containing an array of software programmable gate arrays [Anon1] equivalent to 54000 gates, four general purpose DSP processors and up to 1728 Kbytes of fast static RAM, accessible by the gate arrays as well as by the DSP processors [Enge1, Scho1]. When the board is used for algorithmic or bit-true emulation, the application algorithm will be mapped on the DSP processors, while the gate arrays will implement a fast parallel cross-bar between the processors mutually and between the board and the outside world. When the board is used for architecture-true emulation, software tools will map the application algorithm onto the gate arrays, to which the DSP processors are attached as co-processors.

As a case study, section 1 demonstrates briefly the different steps required to develop an ASIC which implements a digital audio application. This example is too simple to justify the development of a high powered tool like GRAPE but still it can give a clear indication of the different tools that are required. Section 2 presents a discussion of existing development tools for DSP. From this overview, it is clear that none of the available tools can support the complete design path from specification via analysis, simulation and emulation up to implementation on a general purpose multiprocessor or integration on an ASIC. This was the motivation for the plan to develop GRAPE as a shell, which integrates existing tools as much as possible.

The rest of the paper is devoted to the tools required for the simulation and emulation process. In the *specification phase* (section 3), the behavior of the application algorithm is described. This single specification is used as an input for the simulator, emulator and silicon compiler. The *compilation phase* (section 4) generates the code for the DSP multiprocessor. It includes translation of the high-level language modules to relocatable assembly code, assignment of the modules to the processing nodes, scheduling the modules assigned to a node and linking them into executable code. In the *evaluation phase* (section 5), the user verifies if the algorithm is satisfactory, and figures out the minimal word length for each signal in order to select the cheapest DSP processor or to reduce the chip area of the ASIC without violating the frequency domain specifications.

1. DESIGN EXAMPLE: A PRE-AMPLIFIER FOR DIGITAL AUDIO

As an example of an R&D project in the DSP field, we consider the design and implementation of a pre-amplifier for digital audio. This design consists of a cascade chain of a first order offset filter, a graphic equalizer and a third order scratch filter that can be switched on and off. The graphic equalizer is realized by 10 cascaded second order sections (see 'application' in figure 3). A top-down design from specifications to an IC-layout was presented in [VGin1]. A programmable signal processor could also be the target for such an algorithm but in this example, DSPs are only employed as a vehicle for prototyping. We will briefly step through the design flow of this example with emphasis on software tools for improving the design time and quality.

Algorithm Specification, Analysis and Optimization

In a first step, frequency domain specifications such as cut-off frequencies, stop-band attenuation and pass-band ripple, constitute the constraints for the transfer functions of the filters. The order of these transfer functions must be minimal to obtain the cheapest realization with respect to hardware requirements, while still satisfying the frequency domain specifications. The next step is the synthesis of the filter networks, resulting in a block diagram (or signal flow graph) consisting of adders, delay elements and multipliers. While techniques exist to derive these functions from their analog equivalents [Oppe1], direct synthesis of the digital filter from specifications is available from a number of filter design packages.

The word lengths of coefficients and signals still have to be minimized from their virtually infinite precision. The result may influence the selection of the cheapest DSP-processor or save silicon area on a custom chip. Therefore, the filter coefficients must be truncated to minimal word lengths, while still satisfying the frequency domain specifications. On the other hand, the necessary signal word length is determined by finite word length phenomena such as statistical quantization noise, limit cycles bounds, and overflow bounds. Therefore, the optimal word lengths must be determined. To do this the designer can use analysis and bit-true simulation tools.

Real-time Emulation

The optimized digital filter algorithm is in principle ready for implementation. However, for many applications a real-time prototype is required for various reasons: auditive evaluation, proof of concept, marketing approval, tests with real world stimuli, testing of interfaces, etc. Therefore, a bread-boarded prototype, which can be used for the pre-amplifier functions, in combination with application specific digital filters for analog to digital (A/D) and digital to analog (D/A) conversion, is necessary. A functional block diagram of such an existing prototype set-up, is depicted in figure 3. A better solution for this real-time emulation is a multiprocessor configuration, consisting of general purpose DSP-chips, which serves for emulating the audio functions in software. Such a reusable set-up saves a lot of design time compared with application specific bread-boards. A vast amount of processing power is also desirable to provide sufficient experimentation room for additional evaluation

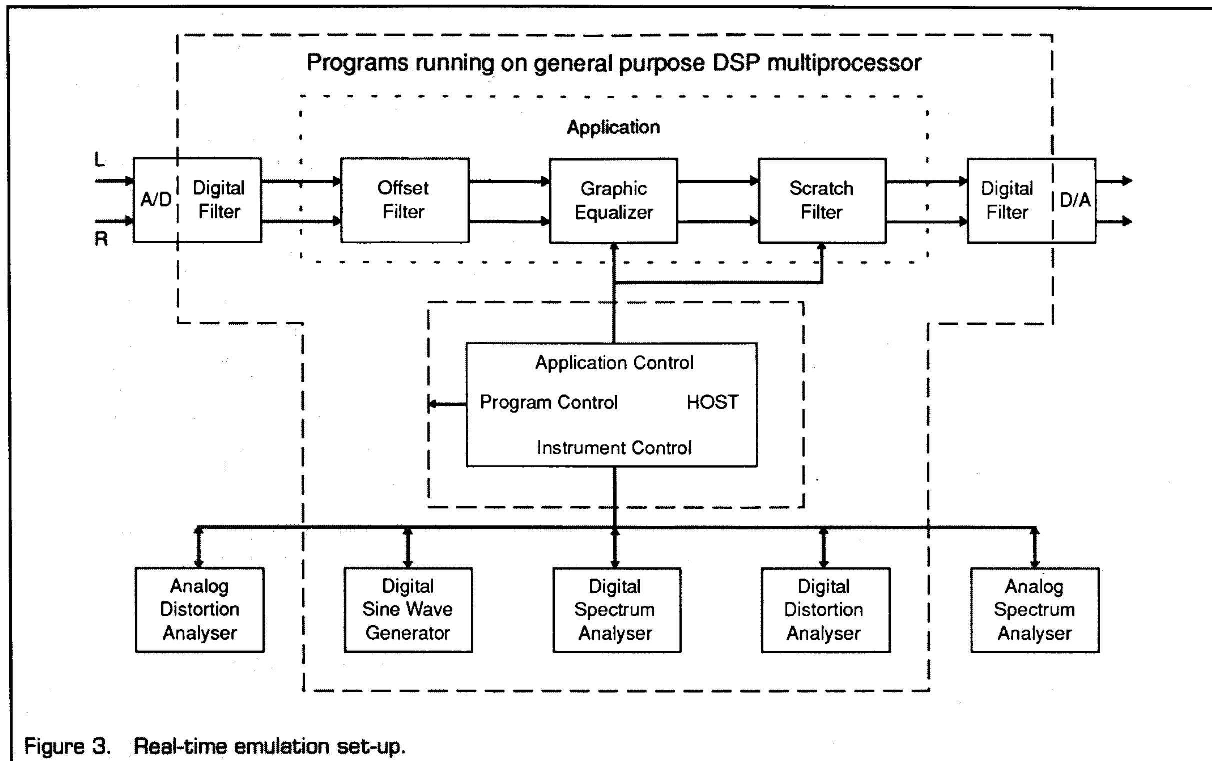


Figure 3. Real-time emulation set-up.

and emulation features such as bit-true modeling of the operations, digital signal preparation and post-processing, over-sampling, distortion analysis, etc.

To use this reusable set-up, code generators, high-level compilers, and function libraries for the DSP-processor, operating on top of the existing traditional development tools are crucial. It is also desirable that monitoring can be performed in a digital way, such as the determining of the signal-to-noise ratio of the filter bank. Monitoring software must be downloadable on the same multiprocessor and serve as virtual instruments. In this way, it can be avoided that the analog parts, which are often the least accurate, influence the measurements. On the other hand, mixed analog/digital measurements such as the characterization of the A/D- and D/A-parts, benefit from a central control by the host. Available instrumentation packages provide virtual panels and displays for the host screen. When the application is programmable such as the graphic equalizer, also those settings can be controlled through such a panel.

Implementation

Once verified, the algorithm can be targeted to its implementation form, a programmable signal processor or a custom chip. For this example, a customized bit-serial implementation has been chosen. A silicon compiler, providing DSP-architecture synthesis on top of general purpose I.C.-layout and verification software, has been used. If the target of the algorithm is a program for a general purpose DSP-processor, a code development tool for this processor is necessary.

2. DISCUSSION OF EXISTING DESIGN TOOLS FOR DSP

Many software tools for assisting DSP-designers have been realized so far. Already in 1979, an important set of programs has been collected by the DSP-committee of the ASSP Society and published by IEEE press [Anon2]. Ever since, the list of software systems, programs, subroutines and algorithms has been growing fast. It is the authors' opinion however, that presently no commercially available integrated CASE tool exists, which covers the total design flow for DSP. For an assessment of the state of the art, we will first give an overview of the partial solutions that are used in the different design phases. After this summary, we will derive some general trends for DSP software and suggest a number of desirable developments.[1]

Functional Families of DSP Tools

In this section we will group the existing software packages that are used in the different design steps,

[1]It is not the intention of the authors to classify specific tools and products or to endorse particular houses. The mentioned names of specific tools are for illustration of the expressed trends only. The reader is urged to contact the vendors or research institutions for up-to-date specifications in this fast moving field.

conforming to the design flow of the example in section 1. Hereby, we will emphasize features and evolutions from a user's viewpoint.

Algorithm specification, analysis and simulation tools

As mentioned in the design example, the tools used in the algorithm specification, analysis, and simulation step can be divided in two classes, namely filter design packages and bit-true simulation tools.

Filter design packages

Tools for digital filter design form perhaps the most mature group of design tools, as they can rely on a lot of research results, rooted in systems theory, numerical analysis, and classical network theory. Software packages such as from the IEEE [Anon2] for classical digital filters and FALCON [Gazs1] for wave digital filters are in fact crucial in turning the rather mathematical digital filter theory into practice for the designers community. Engineering of these software already resulted in user friendly and integrated synthesis modules such as in ILS, MatLab [Litt1, Mole1], MATRIX-X [Shah1], DFDP2, FDAS, and HYPERSIGNAL.

Most of the filter synthesis packages are able to approximate classical filter structures according to e.g. butterworth and elliptic characteristics. Approximation of more arbitrary frequency responses is also possible for magnitude [Stei1] or for both magnitude and phase [VdEn1]. However, these tools try to approximate a nominal frequency response. This is still reminiscent to analog design practices where design centering is desired. Because digital filters are not subject to aging and wear, filter synthesis tools should exploit the design margins between upper and lower limits in a tolerance diagram of the frequency response. Such techniques are, however, not yet reported to the authors' knowledge.

Analysis and bit-true simulation tools

A number of software environments exist that allow for describing DSP algorithms and performing time-domain simulations, and frequency-domain analysis. One approach is block signal oriented. In ILS, for instance, functions are modeled by separate programs in sequential languages, such as FORTRAN, operating on blocks of signals. Feedback loops in the DSP-algorithm must be encapsulated in the programs. The other approach is based on a netlist in terms of DSP primitives such as addition, multiplication, etc. In this way, arbitrary linear algorithms can be described without programming (e.g. in SPW, DSP-DIGEST [Clae1], ODYSSEE [Covi1], and SILAGE [Hilf1]). This description may contain feedback loops and different sampling rates (i.e. multirate system).

An environment such as DSP-DIGEST [Clae1, Clae2], which was used in the design example, gives the possibility of optimizing the length of the filter coefficients and determing the minimum number of nonzero coefficient bits for cheap multiplier-less shift-and-add operations. In, e.g., DSP-DIGEST and the SILAGE language for DSP, different word lengths can be attached to any of the nodes, offering both bit-true simulation and analysis of quantization noise, overflow and limit cycles [Catt1] from the same description.

For nonlinear systems, however, there is a lack of direct analysis tools. Therefore, we are restricted to time-domain verification. To get an idea of the frequency-domain behavior, post-processing algorithms, e.g. FFT, harmonic distortion analysis [VPet1] and signal statistics, are available.

Traditional plotting tools, which are used to display the results of the analyses and simulations, are currently enhanced with electronic spreadsheet like mathematical formulae entry (DADiSP, PCI-SNAP). Integration with simulation and analysis tools will further allow for on-line updates of the displayed curves, which results in a more interactive operation.

Real-time Emulation Tools

The previous discussion stated that two types of design tools are desirable for real-time emulation: a code development system for general-purpose DSP-processors, and an instrumentation package.

Code development systems for general purpose DSP-processors

As in the microprocessor world, the manufacturers of chips and boards make available sets of low-level development tools: assemblers, simulators and debuggers. Time associated with the tedious assembly language programming can be saved by automatic generation of assembly code. This is provided by filter synthesis programs such as FDAS, HYPERSIGNAL, and FALCON. This can result in rather efficient code, because the block diagram is known in advance. Another way to save time, is the use of a high-level language. Recently, C-compilers became available from several DSP-chip vendors. Compilers for the SILAGE language are discussed in [Geni1] and, for bit-true compilation, in this paper. The latter is especially suited for fast prototyping. A retargetable MoDL compiler is described in [Jaco1].

Programming time can also be saved by automating services such as I/O, file-handling, memory allocation and multi-tasking. A tool as SPOX is intended to mask those implementation details from the user by creating a "virtual DSP" model on both the host and the real-time hardware, in combination with a mathematical C-macro library.

Instrumentation packages

During the emulation, interfacing with external instruments is useful for the analysis of real world data and combined digital and analog measurements

[Smal1]. Tools such as ASYST and SPW provide interfaces with GPIB-based measurement setups. In addition, virtual front panels and displays can be emulated on personal computers by packages such as LabView, TestTeam and PCI.

Implementation level tools

For the implementation of the algorithm, two possibilities exist. First, it can be realized on a general purpose DSP-processor. Hereby, the code development systems can be used, that were already discussed in the previous paragraph. Second, the algorithm can lead to an IC-layout. Therefore, a silicon compiler is needed.

Silicon compilers

Provided a sufficiently large market or stringent throughput or size requirements, the DSP-design will be targeted to a semi-custom or full custom I.C.-implementation. Silicon compilers specialized for DSP that assist in the layout and/or hardware architecture synthesis have been developed: (i) FIRST [Deny1] and CATHEDRAL-I [Jain1] for hardwired bit-serial implementation, (ii) LAGER [Pope1] and CATHEDRAL-II [DeMa1] for microcoded bit-parallel implementation and (iii) CATHEDRAL-III [Note1] in development for hardwired bit-parallel implementation. Such CAD-systems are now gradually being introduced for production use in the industry [e.g. Dela1] and will soon play a crucial role for the design of time critical applications.

Because of the overhead in the compiled results of these early tools, it may be still a matter of two to four years before top-down DSP-compilers for customized silicon will be used for the majority of high volume designs which are critical in terms of performance or silicon area.

Trends and Requirements

From the previous discussion, we can conclude that the current DSP design tools are insufficient to offer a user-friendly design environment:

- Filter design packages are lagging behind the development of the theory; most of the currently available one-dimensional filter synthesis packages are shells around programs with a functionality comparable with the IEEE programs [Anon2]. Meanwhile however, new one-dimensional filter structures have been invented such as wave digital filters [Fett1], LDI filters [Brut1] as well as novel filter banks [Vayd1]. Also, adaptive filters have been extensively studied and the field of multidimensional filtering is growing rapidly as well. In automatic tools, however, little of this theory is available.
- Simulations nowadays require extremely long simulation times. Times in the order of one week or even more on supermini computers or engineering workstations are no exception in industry. This is especially true with highly over-sampled systems or analog interfaces. Also, when listening tests are required for evaluation of an algorithm, many operations have to be performed for a few seconds of audio or video.
- Emulation is only possible by the expensive and time consuming method of bread-boarding the system. Neither the software nor the hardware for a reusable emulation set-up are available.
- The implementation of the algorithm requires assembly language programming or manual IC design.
- The different design-tools are only partial solutions. No integrated CASE tool that covers the total design flow for DSP is available.

Nowadays, however, a number of trends can be observed, which promise improvements in this situation:

- Filter design packages that support the automatic design of new filter types are emerging, e.g. FALCON for wave digital filters.
- New hardware accelerators that turn slow hosts into powerful DSP-workstations, are becoming available as uniprocessor plug-in boards and multi-processors are being worked out [Enge1, Enge2, Kok.1].
- Manual coding for DSP-processors is being replaced by compilation from high-level descriptions.
- Integrated tools that blend existing and new tools into larger design tools are becoming available. This was demonstrated with CATHEDRAL-I [Jain1], which resulted in an automated path from frequency domain filter specifications to silicon layout.

To come to a user-friendly DSP design environment, however, some future improvements will be necessary:

- The designer experience and DSP theory must also be captured in synthesis and optimization tools. This must lead to the automatic design of digital filters from arbitrary specifications expressed in a tolerance diagram and to the algorithm synthesis for non-linear functions.
- The code-generation tools for general-purpose DSP-processors must further be improved by optimizing the code generators and compilers. Hereby, bit-true compilers need some special attention. Also, parallel processing support for multi-DSP-processors must be worked out.
- The different tools must be integrated in an extensible CASE system with standard representation, languages, libraries and interfaces.

In the sequel, an experimental version of an integrated system is presented as example: the CASE tool GRAPE for DSP, in development for a personal computer environment. While combining as much as possible available programs, the original developments within GRAPE are concentrating mainly on hitherto missing topics in DSP tool systems related with graphical programming, bit-true emulation and parallel processing support.

3. SPECIFICATION OF DSP ALGORITHMS

Specification languages for DSP algorithms may be divided in three sub-classes: sequential languages, functional languages and graphical techniques [Lauw1] (see also the box on 'Languages for DSP multiprocessors').

The use of a conventional *sequential language* (C, Fortran, Pascal, . . .) may seem the most obvious approach due to its widespread use, to the availability of compilers for almost all target processors and to the large libraries with useful modules that already exist. However, both the ASIC implementation as well as the emulation on a DSP multiprocessor allow for the concurrent execution of multiple tasks. Sequential languages mask the inherent parallelism of the application. A parallelizing compiler is thus needed to extract the data dependency graph out of the sequential program description [Whit1]. The construction of such a compiler is a very complex task and, up to now, they are only able to find part of the implicit parallelism in the program [Coll1].

To be able to extract the data dependency graph automatically by a compiler out of a textual representation of a program, we need a high-level language which forbids the use of constructs masking or introducing data dependencies not implied by the program itself. In short, we need a language that textually describes the data dependency graph. *Functional languages* possess this property [Ager1, Davi1]. Silage, for example, is a functional language that has been designed specifically to describe DSP applications [Hilf1]. One of its distinct features is that it allows the user to specify the word length of each signal. Maybe the only drawback of functional languages is that they still represent the program as a sequential list of statements, although the order in which the statements occur is irrelevant. Indeed, execution order is completely determined by the data dependencies.

The third method is the use of *graphical techniques*, where the data dependency graphs are directly drawn by the system designers. This technique closely resembles the intuitive way they look at the application and think about its behavior. This is especially true for DSP applications where system designers are already used to developing signal flow graphs or block diagrams [Lee.1]. As a consequence, some graphical systems for programming DSP algorithms already exist. Texas Instruments, for instance, offers a graphical environment for programming their Odyssee system, a multiprocessor built up of several digital signal processors of the TMS320xx family [Covi1]. Their graphs represent the z-transform diagrams of the filter algorithms to be implemented. Another example is an internal experiment at Hewlett-Packard, where graphical tools for drawing Nassi-Schneidermann diagrams were offered to various programming pools [Dea.1]. These systems showed, however, that at the lowest level of detail, graphical programming becomes clumsy and dreadful.

From this overview it is clear that a mixed graphical and textual representation of the algorithm is desirable. On the highest levels in programming hierarchy the signal

Languages for DSP Multiprocessors

A first class of languages for DSP multiprocessors is formed by the conventional **sequential languages** (Fortran, Pascal, C, . . .) originally developed for Von Neumann uniprocessors [Gold1]. They possess the common property that instructions are ordered in the same way as they will be executed on such a uniprocessor. During execution, a program counter steps through the list of instructions by simple increments. Thus, non-linear sequencing—for instance with conditional statements, loops, etc.—requires explicit instructions to update the program counter.

To execute a sequential program on a multiprocessor, however, it must be distributed over the processors. Hereby, care must be taken that a processor will not execute an instruction of which one of the operands is not yet computed (e.g. by another processor). The most common method for detecting parallelism and for determining the synchronization points between the processors, is the use of a parallelizing compiler. This compiler analyzes the data dependencies and constructs a data dependency graph [Acke1, Ager1]. The nodes of a data dependency graph indicate the instructions; the edges indicate the data dependency between an instruction that produces a data value and a consuming instruction which needs this data value as an operand. Clever coding of an algorithm in a sequential language, however, obscures the data dependency relationships. As a consequence, most of the parallelizing compilers will only find a small part of the implicit parallelism of a program [Howe1].

By not allowing constructs that obscure data dependencies (like the reuse of temporary variables), **functional languages**, like Silage [Hilf1], make the design of parallelizing compilers easier. In fact, these languages represent the data dependency graph in a textual way. A program consists of a series of equations, each specifying a computation, of which the order is irrelevant. By this, the only ordering imposed on the computation is the one implied by the data dependencies [Trel1].

Instead of describing the data dependency graph in a textual way using a functional language, it is also possible to draw the data dependency graph directly on the screen and to use this **graphical representation** as the programming language. The major advantage of graphical programs is that they are two-dimensional while textual programs are one-dimensional. Therefore, one dimension can be used to indicate parallelism and the other to express the sequencing constraints. The main obstacle against the successful introduction of these graphical programming techniques is the natural conservatism of the programmer. For DSP however, this inertia will soon be ovecome, because system designers are already used to employing signal flow graphs for representing their algorithms [Coll1].

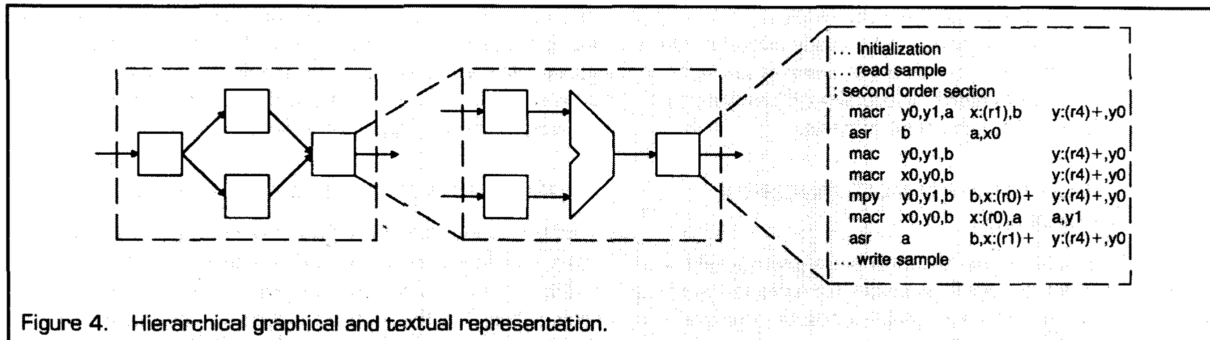


Figure 4. Hierarchical graphical and textual representation.

flow can be represented graphically. This facilitates the compilation towards a multi-DSP processor. On the lowest levels of detail a textual representation avoids clumsy graphics and enables the reuse of the rich software libraries, which are already available for DSP. Therefore, the specification tools of GRAPE support the three types of specification languages. On the highest levels of detail, the signal flow graph is graphically represented by interconnected black boxes. These black boxes may be gradually refined to obtain a hierarchically constructed program tree. The lower level black boxes—the leaves of the program tree—are described in a sequential or functional textual language (Figure 4). The sequential languages C and assembly as well as the functional language Silage are currently supported. Since the point for switching from graphical to textual programming can be chosen freely by the user, pure graphical or pure textual specifications are possible.

To avoid the need to specify the same data more than once during the top-down design of the application, the CASE tool has to provide following properties:

- Hierarchical top-down design in the graphical representation is supported: clicking with the mouse on one of the black boxes opens a separate drawing window for that box in which the signal flow graph can be refined. All information the editor possesses about the black box is automatically inherited by the lower level representation: number of input and output signals, their name, data type, word length, data rate, etc. Extensive cross checking is provided to maintain consistency between all levels of representation and all instances of the same DSP task, so that changes made in one representation are automatically reflected in the others.
- When a (language-sensitive) text editor is called to fill in a black box in some textual language, the program structure and the type declarations for all incoming and outgoing signals can be generated in the language of the users' choice. Here too, cross-checking is required to maintain consistency between the textual and graphical representation of the DSP task.

A few additional tools are necessary to enhance the usefulness of the specification tool:

- *Library toolkit.* The break-through of ASIC's has been made possible by the so-called 'meet-in-the-middle' design philosophy where a number of parametrical standard building blocks, thoroughly tested and carefully optimized, are presented to the system designer. This same 'meet-in-the-middle' philosophy may be employed beneficially in our programming environment by providing the programmer with a library of thoroughly tested and carefully optimized program segments, which he may include in his program as a black box. This black box can contain either a textual, a graphical representation or a complete hierarchical mixed graphical and textual program.
- *Code generator.* A code generator is available for modules that implement a digital filter; after the specification of some parameters including filter type, passband and stop-band ripple and frequency, the filter coefficients are computed and assembly language source code is generated.
- *Icon editor.* Instead of writing the abstract name of the routine in the box representing the task (Figure 5a), it is often more meaningful to fill the box with an icon as depicted in Figure 5b. An icon editor has been included to assist the user in creating meaningful icons for frequently used DSP tasks.
- *Property definitions.* Properties may be assigned to any instance of a DSP task in the form of attributes and to any signal channel. They include data rates, word lengths, real-time constraints, etc. Special provisions have been made to indicate parameters that may be changed during execution of the program. It instructs the compiler to keep track of the memory addresses where these parameters are stored in the target machine. During program execution, it becomes possible to change a parameter (ex. pass-band frequency, amplification, etc.) by just clicking with the mouse on the high-level representation of the module.

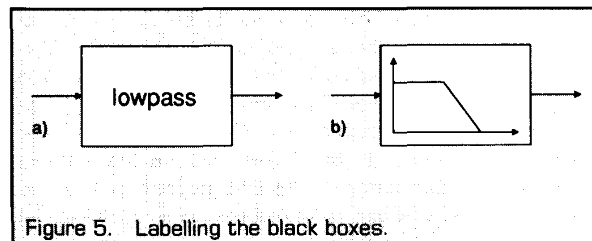


Figure 5. Labelling the black boxes.

All together, the utilities of the specification tool make it possible to describe an application easily and accurately. The single specification can then be used as an input for the simulator, the emulator or the silicon compiler, provided Silage is used for the textual parts.

4. COMPILATION FOR A DSP MULTIPROCESSOR EMULATION FACILITY

Once the modular specification for a given application has been completed, GRAPE provides for tools to transform this DSP algorithm into executable code for the available DSP multiprocessor architecture.

In a first step, each of the modules specified in a high-level language is translated into relocatable assembly code. Currently, GRAPE supports the translation of specifications in C and SILAGE.

The next step is of particular importance, whenever real-time emulation is required. Using estimates of the computation time for each module (extracted by examining the assembly language representation of the modules) and of the inter-module communication requirements, the *partitioner* assigns the modules to the available hardware resources in order to minimize the interprocessor communication. Note that the partitioner has specifically been included in GRAPE to support parallel processing. The tool can be bypassed when code is generated for a uniprocessor.

The next step in the transformation provides for appropriate scheduling of the tasks assigned to a single processing node in the previous step. Besides determining the order in which the tasks assigned to a node have to be executed—this is done by examining the data dependency graph—the scheduler adds communication modules to control the ADCs and DACs and to establish interprocessor communication. Scheduling is followed by a post-optimization step that optimizes the use of the internal processor registers by analyzing the life-time of the variables.

Finally, the individual (relocatable) assembly language modules for each of the processing nodes, are linked together, resulting in the executable code for each of the available processors. In the remainder of this section, the design of a bit-true compiler for the applicative language SILAGE is discussed in more detail. Currently, the compiler produces code for the Motorola 56001 DSP processor [Anon3]. The description of other optimizing compilers for general purpose signal processing may be found in [Geni1] and [Jaco1].

A straightforward—but time consuming—approach to the generation of bit-true code would be to provide an appropriate mask for the result of each computation performed in the generated code. In the compiler described above, a special technique has been applied based on an appropriate representation of the numeric data in the generated program, i.e. by cleverly placing the data in the internal accumulator of the DSP processor, we can avoid the extra overhead required for masking the result of a bit-true computation. Additional instructions ensuring bit-true computation are only incorporated in the assembler program; at the moment an explicit type-conversion—i.e., a change in data word length—is performed in the corresponding Silage program.

5. VERIFICATION OF THE DSP ALGORITHM

At this point the algorithm that must be emulated has been specified and compiled (bit-true or not) for a DSP multiprocessor. The only task left for GRAPE is to provide tools to verify if the algorithm itself is satisfactory and to verify, e.g., the minimal word length for each signal in order to reduce the chip area of the ASIC without violating the distortion specifications.

This verification may cover a number of aspects:

- Verification of the correctness of the developed algorithm (algorithmic emulation). Typically, all the computations in this step are performed using an abundant word length.
- Investigation of the influence of a restricted word length (bit-true emulation). In this step, all the computations in the emulated algorithm are performed with a restricted word length.
- Verification of the real-time behavior of the given algorithm (real-time emulation).
- Verification of the hardware implementation (architecture-true emulation).

To accomplish these verification tasks, the user must be able to

- apply signals to the emulator and to control their characteristics;
- measure the output signals coming from the emulator, to gather statistics of these signals and to compare them with the specifications;
- change parameters of the emulated algorithm.

Three special tools are being integrated in the graphical programming environment GRAPE to accomplish these verification tasks:

1. From within GRAPE, wave-form generators and measuring instruments are directly controlled. The settings of these instruments are manipulated via pop-up front panels and are communicated to them via an IEEE-488 interface bus.
2. A software library of virtual instruments such as wave-form generators, spectrum analyzers, data loggers, distortion analyzers, etc. is available. Indeed, instead of employing separate measuring instruments, it is possible, depending on the signal data rate, to program their function in software and implement these measuring modules together with the application algorithm on the same multiprocessor system. Also, for these virtual instruments, pop-up front panels are used for controlling their parameters.
3. It is possible to design pop-up front panels for the application algorithm that runs on the multiproces-

sor. For example, a front panel may be designed for a digital amplifier that mimics a linear potentiometer. Using these front panels, the user can interactively change the parameters of the algorithm and verify their effect in real-time.

CONCLUSION

This article discussed the usefulness of CASE tools for the design and implementation of stream-oriented real-time DSP applications. The design process of a typical DSP ASIC was analyzed and an indication was given of the development tools that could speed-up the project or improve its quality. A systematic overview of existing development tools was presented next. From this overview, it is clear that no CASE tool exists which covers the whole design process starting from specification via analysis, simulation and real-time emulation up to implementation on a general purpose programmable DSP multiprocessor or integration on silicon. The article then presented the CASE tool GRAPE that aims at filling this gap. This can be achieved by using state-of-the-art techniques for graphical programming, for the code generator, the bit-true compiler, the multiprocessor partitioner, the scheduler, the virtual instrument controller and the silicon compiler. In the near future, GRAPE will be enhanced with tools to enable architecture-true emulation of an ASIC on an array of programmable gate arrays.

REFERENCES

[Acke1] W. Ackerman, "Data Flow Languages," *Computer,* Feb. 1982, pp. 15–25.

[Ager1] Agerwala T., Arvind, "Data Flow Systems. Guest Editors' Introduction," *Computer,* Feb. 1982, pp. 10–13.

[Aho.1] A. Aho, R. Sethi and J. Ullman, "Compilers: Principles, Techniques and Tools," *Addison-Wesley Publ. Co.,* 1986.

[Anon1] Anonymous, "PGA Data Book," *Advanced Micro Devices,* 1988.

[Anon2] Anonymous, "Programs for digital signal processing," Digital Signal Processing Committee, *IEEE Press,* 1979.

[Anon3] Anonymous, "Motorola DSP56000, user's guide," *Motorola Inc.,* 1986.

[Brut] L.T. Bruton, "Low sensitivity digital ladder filters," *IEEE Transactions on Circuits and Systems,* vol. CAS-22, pp. 168–176, 1986.

[Catt1] F. Catthoor, Hugo De Man, and J. Vandewalle, "Simulated-annealing based optimization of coefficient and data word lengths in digital filters," *International Journal of Circuit Theory and Design,* vol. 16, 1988.

[Cara1] M.G. Carasso, J.B.H. Peek and J.P. Sinjou, "The compact disc digital audio system," *Philips Technical Review,* vol. 40, no. 6, pp. 151–155, 1982.

[Clae1] L. Claesen, F. Catthoor, H. De Man, J. Vandewalle, S. Note and K. Mertens, "A CAD environment for the thorough analysis, simulation and characterization of VLSI implementable DSP systems," *Proceedings of IEEE-ICCD'86,* pp. 72–75, Oct. 1986.

[Clae2] L. Claesen e.a., "DIGEST: a digital filter evaluation and simulation tool for MOSVLSI filter implementations," *IEEE Journal of Solid-State Circuits,* vol. SC-19, no. 3, June 1984.

[Coll1] Collesidis R., Dutton T., Fisher J., Metcalf W., "Control of multiprocessor SPS-1000 configurations using principles of data flow architecture," *Signal Processing Systems Inc.,* 223 Crescent Street, Waltham, Massachussetts 02154, USA.

[Covi1] C. Covington et al., "Multiple digital signal processing environment for intelligent signal processing," *Internal Report of the Speech and Image Understanding Laboratory,* Computer Science Centre, Texas Instruments, 1987.

[Davi1] Davis A., Keller R., "Data flow program graphs," *Computer,* Feb. 1982, pp. 26–30.

[Dea.1] R. Dea, V. D'Angelo, "P-Pods: A software graphical design tool," *Hewlett-Packard Journal,* March 1986, pp. 32–35.

[Dela1] A. Delaruelle, "Design of a syndrome generator chip using the PIRAMID design system," *Digest of Techn. Papers, ESSCIRC'88,* Sept. 1988.

[DeMa1] H. De Man, J. Rabaey, P. Six, L. Claesen, "Cathedral-II: A silicon compiler for digital signal processing," *IEEE Design & Test,* Dec. 1986, pp. 13–25.

[Deny1] P.D. Denyer, D. Renshaw, N. Bergmann, "A silicon compiler for VLSI signal processors," *Digest of tech. papers. ESSCIRC'82,* (Brussels, Belgium), pp. 215–218, Sept. 1982.

[Enge1] M. Engels, R. Lauwereins, J. Van Ginderdeuren, "Concept and implementation of a powerful multiprocessor system for digital signal processing," *Internal Report KUL-ESAT 1989,* Jan. 1989.

[Enge2] M. Engels, R. Lauwereins, J. Peperstraete, "Analysis of Interconnection networks for a multiprocessor DSP simulator," *Int. Symp. on Mini- and Microcomputers and their Applications,* Zurich, Switzerland, June 26–29, 1989.

[Fett1] A. Fettweis, "Wave digital filters: theory and practice," *Proc. IEEE,* vol. 74, no. 2, pp. 270–327, Feb. 1986.

[Gazs1] L. Gazsi, "Explicit formulae for lattice wave digital filters," *IEEE Trans. Circuits and Systems,* vol. CAS-32, pp. 68–88, Jan. 1985.

[Geni1] D. Genin, J. De Moortel, D. Desmet and E. Van de Velde, "System design, optimization and intelligent code generation for standard digital signal processors," *Proceedings ISCAS'89,* pp. 565-569, May 1989.

[Gold1] H. Goldstine, J. Von Neumann, "On the principles of large computing machines," *unpublished paper (1946),* included in John von Neumann: Collected Works, Ed. Taub A., Design of Computers, Theory of Automata and Numerical Analysis, Pergamon Press, vol. V, 1963, pp. 1–32.

[Hilf1] P. Hilfinger, "A high-level language and silicon compiler for digital signal processing," *Proc. IEEE CICC Conf.,* May 1985, pp. 213–216.

[Ho..1] Wai H. Ho, Edward A. Lee, David G. Messerschmitt, "High level data flow programming for digital signal processing," *VLSI Signal Processing III,* IEEE Press, Nov. 1988, pp. 385–395.
[Howe1] C. Howe, B. Moxon, "How to program parallel processors," *IEEE Spectrum,* Sept. 1987, pp. 36–41.
[Jaco1] E. Jacobs, "A microcode generator in MoDL for the μPD7720 digital signal processor," *Technical Report 87C014,* University of Twente, The Netherlands, 1987.
[Jain1] R. Jain e.a., "Custom design of a VLSI PCM-FDM transmultiplexer from system specifications to circuit layout using a computer-aided design system," *IEEE JSSC,* vol. SC-21, Feb. 1986.
[Kok.1] W. Kok, A. Yeung, P. Hoang, J. Rabaey, "A multiprocessor system for DSP behavioral simulation," *VLSI Signal Processing III,* IEEE Press, 1988, pp. 295–306.
[Lauw1] R. Lauwereins, "Design of an argument flow parallel computer: from program prganization to multiprocessor architecture," *Ph.D. Diss.,* KUL-ESAT, Feb. 1989.
[Lee.1] E.A. Lee, D.G. Messerschmitt, "Pipeline interleaved programmable DSP's: synchronous data flow programming," *IEEE Trans. on Acoustics, Speech & Signal Processing,* vol. ASSP-35, no. 9, Sept. 1987, pp. 1334–1345.
[Litt1] J.N. Little, "PC-MATLAB, user's guide," *The MathWorks,* Inc., 158 Woodland SL., Sharborn, MA 01770.
[Mole1] C. Moler, "MATLAB user's guide," Dept. of Computer Science, University of New Mexico, 1980.
[Note1] S. Note, J. Van Meerbergen, F. Catthoor, H. De Man, "Automated synthesis of a high speed CORDIC algorithm with the CATHEDRAL-III compilation system," *Proceedings IEEE ISCAS'88,* June 1988.
[Oppe1] A.V. Oppenheim and R.W. Shafer, "Digital signal processing," *Prentice Hall Inc.,* New Jersey, 1975.
[Pope1] S. Pope, J. Rabaey and R.W. Brodersen, "Automated design of signal processors using macrocells," *VLSI Signal Processing,* IEEE Press, pp. 239–251, 1984.
[Sche1] C. Scheers, "User manual for the S2C Silage to C Compiler," *Internal Report,* IMEC, Leuven, Belgium, Nov. 1988.
[Scho1] K. Schoofs, Ch. Verheirstraeten, "Design and implementation of a flexible DSP multiprocessor board using programmable gate arrays," *M.Sc. Diss.,* KUL-ESAT Sept. 1989 (in Dutch).
[Shah1] S. Shah e.a., "MATRIXx: A model building, nonlinear simulation and control design program," in *Computer-Aided Control Systems Engineering* (M. Jamshidi e.a eds.), North-Holland Publishing, pp. 181–207, 1985.
[Smal1] C.H. Small, "Virtual instruments," *EDN,* pp. 121–128, Sept. 1988.
[Stei1] K. Steiglitz, "Computer-aided design of recursive digital filters," *IEEE Trans. Audio Electroacoustics,* vol. AU-18, pp. 123–129, June 1970.
[Trel1] P. Treleaven, I. Gouveia Lima, "Fifth generation computers," *Computer Physics Communications,* North Holland Publishing Company, vol. 26, 1982, pp. 277–283.
[Vayd1] P. P. Vaidyanathan, "A tutorial on multirate digital filter banks," *Proceedings ISCAS'88,* pp. 2241–2248.
[VGin1] J. Van Ginderdeuren, H. De Man, B. De Loore, H. Vanden Wijngaert, A. Delaruelle, G. Van Den Audenaerde, "A High Quality Digital Audio Filter Set Designed by Silicon Compiler CATHEDRAL-1," *Journal of Solid-State Circuits,* vol. SC-21, pp. 1062–1075, Dec. 1986.
[VPet1] P. Van Petegem, P. Vandeloo and W. Sansen, "Efficient least-mean-squares algorithm performs audio distortion analysis on sampled waveform," *Proceedings of 75th AES Convention,* Paris, March 1984.
[VdEn1] A.W.M. van den Enden and G.A.L. Leenknegt, "Design of optimal IIR filters with arbitrary amplitude and phase requirements," *Philips Research Internal Report 6090,* 1986.
[Whit1] Whitelock P., "A conventional language for data flow computation," M.Sc. Diss., Department of Computer Science, University of Manchester, UK, Oct. 1978.

Rudy Lauwereins received the degree in electrical engineering from the Katholieke Universiteit Leuven in 1983 and a Ph.D. in Applied Sciences in February 1989. He is currently a senior research assistant at the ESAT laboratory of the Katholieke Universiteit Leuven in Belgium. His main research interest is the large domain of parallel processing, including fault-tolerant computations and real-time signal processing. His work is sponsored by the Belgian National Fund for Scientific Research.

Marc Engels received the electrical engineering degree from the Katholieke Universiteit Leuven in Belgium in 1988. He joined the ESAT laboratory as a research assistant, where he is currently working towards a Ph.D. degree. His main activity is the design of reconfigurable multiprocessor networks; he is also involved in the development of a case tool for these multiprocessors. His work is sponsored by the Belgian Institute for Scientific Research in Industry and Agriculture.

Jean Peperstraete received the degree in Electrical Engineering, Industrial Engineering and a Ph.D. from the Katholieke Universiteit Leuven, Belgium, in 1964, 1968 and 1973 respectively. He was a design engineer and group leader at Philips Industries and is now Professor of Digital Electronic Systems at the Katholieke Universiteit Leuven. His areas of interest include Computer Architectures, Transputers, data flow and parallel computers.

Eric Steegmans has been the first assistant at the Computer Science Department of the Katholieke Universiteit Leuven since October 1985. His research activities include compiler construction, formal specifications and software engineering techniques. He lectures on software engineering, database management and artificial intelligence in postgraduate courses at the Katholieke Universiteit Leuven and Limburg, U.C. Both his engineering degrees in Computer Sciences (1978) and his Ph.D. are from the Katholieke Universiteit Leuven. From 1978 to 1985 he was engaged in several research projects at the Computer Science Department of the Katholieke Universiteit Leuven.

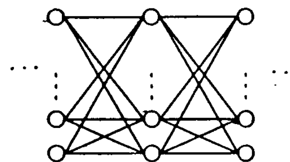
Johan Van Ginderdeuren received the electrical engineering degree from the Katholieke Universiteit Leuven, Belgium, in 1980. During the summer of 1980, he was on leave at Philips Research Laboratories, Eindhoven, The Netherlands. From autumn 1980 to 1983 he obtained an IWONL grant that allowed him to work as a research assistant at the ESAT Laboratory of the Katholieke Universiteit Leuven. In 1983 he was appointed as research assistant under the ESPRIT 97 (Advanced Algorithms and Architectures for DSP) project on the CATHEDRAL system at the Katholieke Universiteit Leuven and the Inter-University Micro Electronics Center (IMEC), Leuven, respectively. In 1986, he joined Philips Industrial Activities in Leuven, Belgium. His current interests are in computer-aided design and I.C.-implementations strategies for DSP, which he has applied in several digital audio applications. He is also leading a campus-liaison group of PHILIPS stationed at IMEC.

TRADEMARKS

ILS	is a trademark of Signal Technology, Inc.
MATRIX-X	is a trademark of Integrated Systems, Inc.
DFDP2	is a trademark of Atlanta Signal Processors, Inc.
FDAS	is a trademark of Momentum Data Systems
HYPERSIGNAL	is a trademark of Hyperception, Inc.
SPW	is a trademark of Comdisco Systems, Inc.
PCI-SNAP	is a trademark of Siemens
DADISP	is a trademark of DSP Development Corporation
SPOX	is a trademark of Spectron Micro Systems
ASYST	is a trademark of Asyst Software Technologies, Inc.
TESTTEAM	is a trademark of Philips-Fluke
LABVIEW	is a trademark of National Instruments Corporation

NEURAL NETWORKS
1989
8 New Tutorial Videotapes
presented by
IEEE's Educational Activities Board
in Cooperation with the
IEEE Neural Network Committee

- ADAPTIVE PATTERN RECOGNITION - Leon Cooper
- NEUROBIOLOGY REVIEW - 1989 - Walter Freeman
- ADAPTIVE SENSORY - MOTOR CONTROL - Stephen Grossberg
- NEURAL NETWORKS: ALGORITHMS AND MICROHARDWARE - John Hopfield
- VLSI TECHNOLOGY AND NEURAL NETWORK CHIPS - Lawrence Jackel
- OPTICAL NEUROCOMPUTERS - 1989 - Demetri Psaltis
- STARTING A HI-TECH COMPANY - Peter Wallace
- REINFORCEMENT LEARNING - Ronald Williams



For more information, call IEEE,
Educational Activities at (201) 562-5499,
or write IEEE, Educational Activities
445 Hoes Lane, PO Box 1331
Piscataway, NJ 08855-1331.

Introduction to

DIGITAL SPEECH PROCESSING

Developed by Andrew Sekey

A smooth transition from the fundamentals of speech production to modern applications!

IEEE MEMBER PRICE $249.00, List Price $695.00

ILP includes study guide, audiotape, textbook, *Digital Processing of Speech Signals*, Lawrence R. Rabiner and Ronald W. Schafer, Prentice-Hall, 1978, and an IEEE Press Book, *Speech Analysis*, ed. Ronald W. Schafer and John D. Markel, IEEE, 1979.

For more information call:
(201) 562-5498, or write

Theresa M. Kirby
Educational Activities
IEEE
P.O. Box 1331
445 Hoes Lane
Piscataway, NJ 08855-1331