An Algorithm Design Environment for Signal Processing

RLE Technical Report No. 549

December 1989

Michele Mae Covell

Research Laboratory of Electronics
Massachusetts Institute of Technology
Cambridge, MA 02139 USA

This work was supported in part by the Defense Advanced Research Projects Agency monitored by the Office of Naval Research under Grant No. N00014-89-J-1489, in part by the National Science Foundation under Grant No. MIP 87-14969, and in part by Sanders Associates, Incorporated.

An Algorithm Design Environment for Signal Processing

by

Michele Mae Covell

Submitted to the Department of Electrical Engineering and Computer Science
on August 11, 1989, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The computer is used in signal processing primarily for numerical calculations. Recently, a number of signal-processing environments have evolved which simplify the initial creation of a signal-processing algorithm from a series of low-level signal-processing blocks. Despite this progress, the majority of the design process is generally completed without computer support: the analyses of the properties of the selected algorithm are generally completed by hand as is the manipulation of the algorithm to find efficient, input/output equivalent implementations. This thesis explores some of the issues involved providing software tools for the symbolic analysis and rearrangement of signal-processing algorithms as well as for the initial algorithm selection.

A software environment is developed, supporting numeric signal-processing computations as well as symbolic analysis and manipulation of signal-processing expressions. The areas in which this thesis contribute lie primarily in the symbolic manipulation of signal-processing expressions. To allow for the efficient manipulation of a variety of "regular" algorithms, such as polyphase and FFT structures, correspondence constraints are introduced and used to guide the rearrangement of these structures. Detailed cost descriptors are developed to allow the environment to accurately compare the costs of the various equivalent implementations: these comparisons are used to reduce the number of implementations presented to the user by removing the uncomputable and the computationally inefficient forms.

The potential of constrained algorithm manipulation is demonstrated using two examples. The problem of non-integer sampling rate conversion is briefly considered. The more complex problem of detecting and discriminating FSK codes in sonar returns is explored in detail. A third example, on the recovery of in-phase and quadrature samples of an RF signal, is used to highlight some of the limitations of the design tools developed in this thesis.

Thesis Supervisor: Alan V. Oppenheim
Title: Professor, Department of Electrical Engineering and Computer Science

Dedication

Many of my colleagues have brightened my days with their friendship during the course of this thesis. Jerry Roylance and Jean-Pierre Schott have helped me keep my sanity as well as contributing to my insanity. Lori Lamel and Tae Joo have given me their friendship and support. Roz Picard and Deborah Gage have provided me with examples of personal generosity which I have tried to imitate. Cory Myers has been a good friend as well as an advisor and a colleague. John Hardwick, Dan Cobra and Jeff Rodriguez have been my friends as well as my office-mates. Joe Bondaryk has been and will always be my "fellow TA." My friendship with Joe, with John Richardson and with Steve Isabelle has had the added dimension of a shared enthusiasm for jazz. Greg Wornell and Jim Preisig have brightened my days with both friendship and humor.

My days of graduate work have also been brightened by the love and the understanding of my grandparents, my Dad, and Carole.

This thesis is dedicated to all these people and, most especially, it is dedicated to my mother and my grandmother, who prompted my entrance into graduate school.

Acknowledgments

Professor Alan Oppenheim has been a strong proponent of algorithm design environments for signal processing. I would like to thank him for his guidance, his encouragement and his support of this work. Over the course of the last five years, Al has inspired my respect for his abilities as a teacher and as an advisor. I hope that both our friendship and our professional interactions will continue beyond the end of this thesis.

My sincere thanks go to Professor Hal Abelson. His detailed and important comments on the first draft are reflected in the current organization and emphasis of this thesis.

Professor Bruce Musicus has also given his time and his guidance to improving the research reported here. The meetings which he organized with Cory Myers, Dennis Fogg and Prasanna and his input at those meetings have had a direct impact on the direction of my research. These meetings and his comments on the content of my thesis have helped me complete this work and his friendliness and good humor have made the task more enjoyable.

Cory Myers has provided both guidance and support throughout this thesis. Cory has helped me with difficult technical problems through his willingness to listen and through his enthusiasm and his talent for finding solutions. Just as important to me has been Cory's friendship: this friendship has helped me through many difficulties as surely as has Cory's technical advice.

My thanks also go to John Richardson and Gary Kopec. The match between John's research and my research is surprising in its apparent perfection. Gary Kopec provided both his time and his advice early in this research.

The National Science Foundation has generously provided support for part of this research, through their graduate fellowship program, as have the Advanced Research Projects Agency and Sanders Associates, Inc.

Contents

Contents

Chapter 1

Introduction

In carrying out signal-processing system design, there are some design tools currently available (Morris, 1977; Gethöffer, 1980; Dove et al., 1984; Johnson, 1984; Bentz, 1985; Chefitz, 1987; Hicks, 1988; Nagy, 1988), but they primarily provide a convenient environment for writing programs. There is no environment generally available which remaps signal-processing algorithms specified at a high level of abstraction to algorithmic descriptions that are more computationally efficient. Accomplishing this requires non-trivial, global transformations of the problem statement in order to work towards an efficient low-level solution. The focus of this thesis is the symbolic manipulation of signal-processing algorithms to find efficient implementations or, more accurately, a software environment which supports and partially automates these manipulations. A successful design environment could, in some cases, improve on designs generated by experienced engineers. A more likely and also highly desirable outcome is an environment which quickly produces reasonably efficient designs of signal-processing algorithms.

This chapter introduces the thesis topic, an environment for digital signal processing algorithm design, by tracing out the long-term goal of research in this area and briefly considering some of the previous and ongoing research in this and related fields.

1.1 Digital Signal Processing Design Environment

There are many examples in which reconfiguring an algorithm through a large number of straightforward transformations can lead to major gains in performance. Multirate techniques for signal processing which interchange various levels of downsampling and filtering can vary by one to two orders of magnitude in computational requirements, depending on the specific ordering of the successive stages (Crochiere and Rabiner, 1975). Similarly, in many digital filtering applications with fixed-point arithmetic, the effects of arithmetic roundoff errors in the resulting signal-to-noise ratios of the processor output are highly dependent on the ordering of sections (Jackson, 1970; Chan and Rabiner, 1973a; Chan and Rabiner, 1973b; Jackson, 1986). Clearly this type of transformation could and should be handled in a semi-automated design environment.

Other examples in which significant improvement can be expected through algorithmic transformation are the mapping of algorithms onto highly pipelined and parallel architectures. For systolic or single-instruction, multiple-data (SIMD) arrays, for example, it is well known that straightforward approaches to multiplying matrices or solving linear equations lead to inefficient programs requiring the broadcast of data across the array and resulting in the poor utilization of the processor resources. Techniques are known, however, for using staggered sweeps of processors through the geometrically regular data flow-graphs which define these problems, to achieve extremely efficient utilization of the array (Barnwell et al., 1982; Barnwell and Schwartz, 1983; Schwartz, 1985). Unfortunately, such sweeps require unusual patterns of data storage, data access, communication, and operation timing. The success of such inexpensive, massively parallel architectures might well depend on the development of high-level compilers capable of rearranging the computation into forms suitable for such machines, and capable of choosing appropriate data-sweep patterns through each step of the computation to maximize the storage, computation and communication efficiency of the machine.

These examples illustrate the application area which digital signal processing design environments may affect. Specifically, research in this field is ultimately aimed at supporting the process of signal-processing design from the initial conception of the system and analysis of its characteristics through the global rearrangement of the algorithm down to the selection of a particular processor architecture and the scheduling onto that hardware. Although this particular thesis addresses more modest goals, its contributions in the area of algorithm design represent progress along a path toward this ultimate goal.

In deciding to attack the problem of signal-processing algorithm manipulation, there are two motivations: the benefit to the field of digital signal processing and, as a possible bonus, the further exploration of the AI paradigms and their interactions. The field of DSP can benefit from semi-automatic algorithm manipulation both from the reduction in detail that the system designer must consider and also from the discovery of novel implementations of common signal-processing operations. One may argue that the second benefit will seldom be seen, that very few new DSP implementations can be found using rudimentary manipulations. A counter argument is that, even in the short time over which research in this area has extended, novel implementations in well-developed areas of signal processing have been found (Myers, 1986; this thesis). Even ignoring this possible benefit, the field of signal processing can still benefit from environments like the one researched here in much the same way that fields using integral calculus benefit from MACSYMA (Rand, 1984). MACSYMA has not resulted in the discovery of new closed-form solutions to integral equations.[1] Instead, the contribution of MACSYMA has been to relieve the user of the burden of knowing about and completing the mathematical transformations which are embedded in it. Similarly, a design environment like the one proposed here can partially relieve the user of the burden of rearranging algorithms by hand.

The field of digital signal processing also has a fairly unique combination of characteristics which allows the exploration of interactions between AI paradigms. Since the included signal-processing manipulations are exact as opposed to heuristic, algorithm manipulation can avoid the issues of approximation and certainty measures. The

[1]It is interesting to note that MACSYMA has, indirectly, resulted in the derivation of new numerical methods, due to a heightened interest in this field.

manipulation of signal-processing algorithms has a high branching factor, making the use of higher-level rules, like control strategies and search-space reduction, an important consideration. Another interesting characteristic of signal processing is its largely object-oriented structure: signals and systems are the primary focus of signal-processing manipulation; each signal and system has a separate identity; each can be classified according to its properties; and the behavior of each is most easily described according to this classification. Thus, the field of digital signal processing can be used as a natural avenue for exploring interactions between object-oriented programming and rule-based systems.

To determine our chance of success in this area of research, a goal must be agreed upon. The goal that is being set for the short term is more modest than the long-term goal described earlier. The goal of this dissertation is to further demonstrate the potential of algorithm manipulation in digital signal processing, by exploiting some of the characteristics of the field of DSP and by combining some of the current AI paradigms. Although this goal is modest compared to our stated long-term goal, it is a necessary step on the road to a practical implementation of a signal-processing design environment: the tools that are appropriate for signal-processing algorithm manipulation must be found before we can hope to further assess the practicality of semi-automatic algorithm design. Some of the evidence that leads us to believe that success is possible in our long-term goal is the previous success of E-SPLICE (Myers, 1986) which used only basic AI tools and our own success with ADE, which is described in detail in this thesis.

1.2 Primary Contributions

The primary contributions of this thesis lie in the area of algorithm reconfiguration. One of the stated goals of a signal-processing design environment is the provision of semi-automated algorithm design. This capability is provided in the Algorithm Design Environment (ADE) via the enumeration and partial ranking of input/output equivalent implementations of an algorithm. At the user's request, the design environment searches for implementations of a signal-processing expression. To obtain equivalent implementations, the environment applies algorithmic transformations to the signal-processing expression and to all its component subexpressions. In addition, the transformed algorithms are themselves used as seeds for further transformations. To avoid combinatorial growth of the search space, due to the independent manipulation of component subexpressions, this thesis introduces the concept of correspondence constraints. Correspondence constraints are used to reflect and maintain the internal regularity of algorithms, such as polyphase and FFT structures. By enforcing these constraints, the size of the search space is reduced from $\mathcal{O}(M^N)$ to $\mathcal{O}(M)$, where N is the number of parallel subexpressions being constrained and M is the number of equivalent implementations which are uncovered for each of these subexpressions.

To allow equivalent implementations to be compared, this thesis also develops a detailed cost metric for describing the computational requirements of an implementation. These cost metrics are not used in actually reducing the size of the search space, since the costs of separate subexpressions can interact and since the recursive search used in finding equivalent expressions allows a simple, local change to have a global effect on the implementation. Instead, the cost measures are used as a filter for removing uncomputable or inefficient implementations of a signal-processing expression before the equivalent implementations are presented to the user. As such, the cost metric must provide an accurate reflection of the cost of each implementation. Vectors of operation counts and memory requirements provide the basis for describing the cost of each algorithm: vectors must be used since the actual time and area cost of any algorithm is highly dependent not only on the algorithm, but also on the selection for the hardware architecture and the scheduling onto that hardware. Index dependencies are also explicitly included in the cost metrics: thus, the difference between a signal adder and an FFT would be reflected in the size of the computational block which each cost vector describes and in the indices over which the costs are imposed. Using these index-dependent cost vectors, the actual cost of each implementation can usually be accurately described.

Another important contribution of this work is the development of signal and system representations which allow information to be easily and efficiently shared between related objects. The idea of abstract objects, first introduced by Myers (1986), allows sets of signals and systems to be manipulated simultaneously. For example, by using the description "a real, symmetric discrete-time sequence" to characterize a signal, the set of all real, symmetric discrete-time sequences can, in effect, be manipulated simultaneously. Due to the generality of this description, the result of invoking the description "a real, symmetric discrete-time sequence" is a distinct abstract sequence for each invocation. If the result were identical on each invocation, two distinct abstract sequences, both characterized by a single description, could not be considered simultaneously. This results in the generation of multiple, distinct but closely related objects. In particular, all of the information which is known and which can be determined about the instances of a single abstract description is identical, to within a simple substitution. This thesis introduces a two-level representation for abstract objects to allow this common information to be shared. A similar two-level representation is developed for signals and systems which depend on an abstract object. The advantage of these two-level representations is the ease with which information derived for one instance can be reused in characterizing a related instance.

In this thesis, the problem of FSK-code detection and discrimination is used to illustrate many of the above issues. The detection and discrimination of FSK-codes in a sonar environment is completed in two largely separate stages: matched filters are used to detect the individual frequency chips which make up the FSK-code signals and incoherent summation is used to form the complete FSK-code detectors from these filtered signals. The size of the unconstrained design space for this problem highlights the necessity for regularity constraints in algorithm manipulation. Innovative implementations for the matched filters are uncovered for three alternate frequency-chip shapes. These implementations and the paths by which they are obtained are discussed in detail in

Chapters 3.

The notational conventions used in this thesis are discussed in the next section of this chapter. The chapter then closes with an outline of the thesis and an index to the signal-processing design examples within the thesis.

1.3 Notational Conventions

Throughout this thesis, the notational conventions of Oppenheim and Schafer (1989) are used. For example, $x[n]$ is used to represent a discrete-time sequence; $X(e^{j\omega})$ represents its Fourier transform; and $X(z)$ represents its z transform. Downsampling $x[n]$ by a factor of N, represented graphically in Figure 1-1-a, removes $N-1$ of every N samples, resulting in $y[n] = x[Nn]$. Upsampling $x[n]$ by a factor of N, represented graphically in Figure 1-1-b, inserts $N-1$ zeroes between samples, resulting in $y[n] = \begin{cases} x[n/N] & n = kN \\ 0 & otherwise \end{cases}$. Figure 1-1-c shows the graphical representation of the convolution of $x[n]$ and $h[n]$. Figure 1-1-d through 1-1-f show alternate graphical representations of shift operations. Figure 1-1-d shows a single-sample delay, so that $y[n] = x[n-1]$. Figure 1-1-e and 1-1-f both represent $y[n] = x[n+k]$ graphically.

In addition, LISP descriptions and objects are represented within this thesis. Small capital letters are used for LISP inputs and outputs. Thus, (OUTPUT-OF (SHIFT 5) (IMPULSE-SEQUENCE)) represents a request for the sequence $\delta[n+5]$. As is customary, LISP variables or symbols are represented using their names and LISP lists are represented by enclosing their components by parentheses. Object-based representations, such as will be used for signals and systems, are generally represented by enclosing the LISP expression which generated them in braces: thus, the actual sequence $\delta[n+5]$ will be represented as #<(OUTPUT-OF (SHIFT 5) (IMPULSE-SEQUENCE))>. Evaluating (NAMED-SETQ *var obj*), when *var* is a symbol and *obj* gives an object-based representation, binds the symbol *var* to *obj* and changes the printed representation of *obj* to be #<*var*>. For example, evaluating (NAMED-SETQ X (OUTPUT-OF (SHIFT 5) (IMPULSE-SEQUENCE))) binds

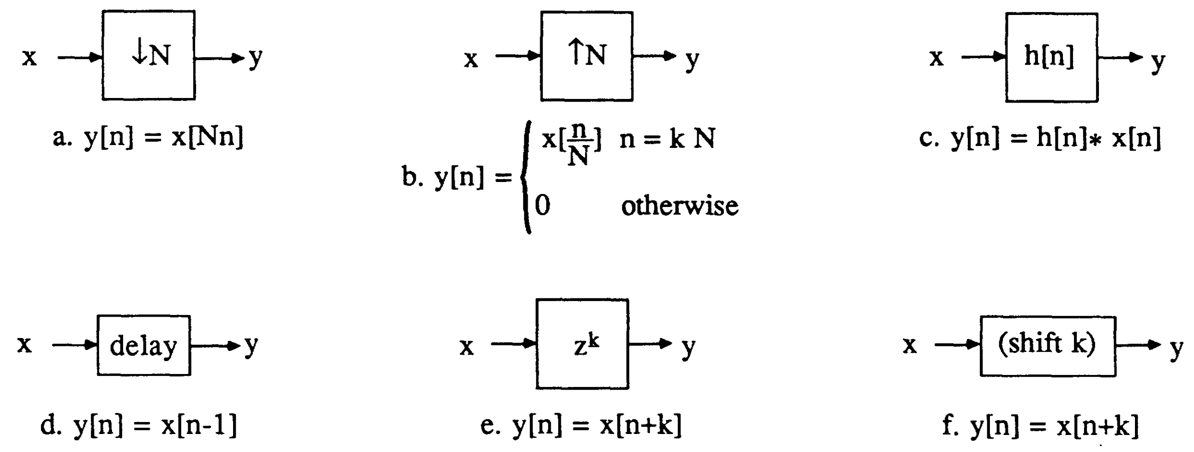


Figure 1-1: Graphical representations of some signal-processing operations

x to the object #<(OUTPUT-OF (SHIFT 5) (IMPULSE-SEQUENCE))> and changes the printed representation of the shifted impulse to #<x>.

1.4 Dissertation Outline

The presentation of the material in this thesis can be separated into two interleaved parts. Material of general interest in signal processing is presented in this chapter and in Chapters 2, 3 and 8. Material on the underlying in an algorithm design environment is presented in Chapters 4 through 7. This organization is the result of an effort to move material of general interest to the beginning of the thesis without having to repeat the discussion of the supporting representations: concrete examples of the use of a design environment are presented before the representational intricacies are discussed.

Chapter 2 reviews some the previous and concurrent research into languages and environments for signal processing. Chapter 3 introduces an FSK-code sonar detection problem which will be used throughout this thesis to illustrate the issues under discussion. Chapter 3 also introduces the Algorithm Design Environment (ADE). ADE is based on the representational and implementational choices which are advocated within this thesis. Finally, Chapter 3 presents some of the algorithmic structures which were uncovered using constrained search in ADE.

Chapter 4 explores the desired characteristics of the signal and system representations within a design environment. Chapter 5 describes some aspects of a control structure having both the flexibility required for general signal-processing design and the efficiency made necessary by the size of the design spaces under consideration. Chapter 6 considers the search space for algorithm design and the use of regularity to limit that space. Finally, Chapter 7 develops a cost measure for comparing alternate implementations of an expression.

This thesis closes with Chapter 8 which highlights the strengths and weaknesses of the design environment and offers suggestions for future research in the field of algorithm design environments.

1.5 An Index to the Signal-Processing Design Examples

As mentioned before, the organization of this thesis attempts to present material of general signal-processing interest in the early chapters. As a result, the examples of signal-processing algorithm design are scattered throughout the thesis, instead of being collected into a single chapter.

The problem of designing a bank of matched filters for the individual frequency chips of the FSK-code signal is considered in section 3.4 of Chapter 3. This section includes the design of matched filter banks for three alternate frequency-chip windows.

The implementation of a 4:5 non-integer sampling rate conversion is briefly discussed in section 6.4 of Chapter 6.

Finally, the recovery of in-phase and quadrature samples of an RF signal is considered in section 8.2 of Chapter 8.

Chapter 2

Background

As stated in the previous chapter, the eventual goal of research in the field of signal-processing design environments is to provide the engineer with an environment that will support and expedite all stages of the signal-processing design: the initial selection of a prototypical algorithm; the manipulation and analysis of the algorithm; the exploration of input/output equivalent implementations; and the selection of and scheduling onto a processor architecture.

Many languages and software tools have been suggested to support the first stage of the design process, namely, the selection and numerical characterization of a prototypical algorithm (Morris, 1977; Gethöffer et al., 1979; Dove et al., 1984; Johnson, 1984; Bentz, 1985; Chefitz, 1987; Hicks, 1988). Most of these languages were developed to only support the selection of a particular, computable implementation: as a result, they can only represent a limited subset of the systems fundamental to digital signal processing. For example, they can not represent the discrete-time Fourier transform or the z transform of a general sequence. Furthermore, the majority of these languages can only represent a limited subset of the signals fundamental to signal processing. For example, with the exception of SPLICE (Dove et al., 1984; Myers, 1986) and D-PICT (Hicks, 1988), none of the current languages can accurately represent a general exponential sequence, one of the signals fundamental to transform analysis.

Research into the automatic or the semi-automatic exploration of input/output equivalent implementations has been limited and recent. Myers (1986) developed a symbolic signal representation and some symbolic manipulations of signals on top of a basic signal-processing package. These symbolic manipulations include the capability to analyze signal properties such as non-zero support, period, sample type and symmetry; to rearrange block diagrams of systems without affecting their input/output characteristics; and to partially characterize the computational cost of alternate implementations. Fogg (1988) proposes to explore the manipulation of irregular signal flow-graphs to generate custom VLSI hardware implementations. While Fogg (1988) does not propose to analyze the properties of the system being manipulated or those of the signals flowing through the system, Fogg (1988) does provide suggestions on one of the issues central to this thesis: namely, ways of limiting and guiding the search for "efficient" implementations.

The final stage of the design process which should be supported by the design environment is the selection and scheduling onto a particular processor architecture. Many compilers exist for mapping a fixed signal flow graph onto a fixed set of architectures (Siskind et al., 1984; Traub, 1986; Lam, 1987; Smith, 1987; Zissman et al., 1987). All of these avoid the issue of the architecture selection by assuming that this selection has already been made. Prasanna (1988) proposes to study this joint selection and scheduling process using simple, highly regular algorithms such as the matrix/vector multiply of a discrete Fourier transform.

This chapter examines some of the previous and concurrent work in the areas of signal-processing languages and environments. Environments without the ability to manipulate signals, their properties and their generating expressions are categorized as numeric signal-processing environments, since all of these abilities are generally necessary for symbolic manipulation of signal-processing expressions. Using this criteria, only the research by Myers (1986) has previously resulted in an environment for signal-expression manipulation. A description of the work proposed by Fogg (1988) is included in the section on signal-expression manipulation environments, even though his proposed environment does not meet all the criteria: in particular, no explicit signal or system representation is proposed nor will property manipulation be supported. Instead, this categorization of Fogg (1988) is meant to reflect its emphasis on symbolic manipulation of algorithmic forms. In all these overviews, an effort will be made to summarize the contributions and shortcomings of these pieces of work from the point of view of signal-expression manipulation. The remainder of this section provides an outline of the desired properties of the signal representations, properties which are considered necessary for a complete representation.

Signals in signal processing are express entities (Kopec, 1980): they are not just an ordered collection of sample values, but instead have a distinct identity and inherent properties of their own. Many of their properties, such as non-zero support, domain and symmetry, are closely tied to the sample values, but others, like the cost of computing the sample values between $-\infty$ and ∞, can not be derived from the sample values. Therefore, an explicit signal representation, distinct from a simple ordered set of sample values, is required.

The domain of a signal determines where the signal is defined: discrete-time sequences are defined on all integer time indices and undefined elsewhere; discrete-time Fourier-transform signals are continuously defined on all real frequency indices; and z-transform signals are defined on the annulus of complex indices inside its region of convergence and undefined elsewhere. Any sample value within the defined domain of the signal should be accessible. Accessing a signal inside its domain but outside its non-zero support should return the sample value of the signal at that point, namely zero. This results in an explicit separation of the domain and the non-zero support of a signal, as advocated by Dove et al. (1984) and Myers (1986).

Mathematically, signals are immutable objects: their identity and their properties are fixed and unchanging. For example, the sample values, symmetry and non-zero support of the complex-exponential sequence, $e^{-j\frac{3\pi}{16}n}$, are completely defined and immutable. Using this sequence as input to a system, like an FIR filter, does not alter the sequence but instead produces a new sequence. As pointed out by Kopec (1980), this immutability in signals also simplifies and clarifies the signal-processing algorithms which use them: immutability makes signals referentially transparent.

As demonstrated in Myers (1986), a final ability that is highly useful in signal-processing algorithm design is the ability to generate and manipulate abstract signals, in much the same way as specific signals. For example, various system characteristics can be examined by passing through the system a signal representing some set of signals, like an abstract discrete-time sequence.

In summary, the characteristics which will be considered essential to a complete signal representation include: a distinct signal identity; the ability to manipulate signal properties; an explicit signal domain, distinct from its non-zero support; and immutability. Another highly useful characteristic is the ability to generate and manipulate signal representations corresponding to a class or set of signals.

2.1 Numeric Signal-Processing Environments

This section surveys some of the available signal-processing software used for numerical manipulation of signals. This survey does not attempt to exhaustively catalogue currently available software. Instead an attempt is made to examine the range of signal and system representations.

2.1.1 Array-based signal representations

The most commonly used signal representation is an array-based signal representation. For example, an FIR filter would be represented by an array containing its coefficients and a system to add two sequences would simply add corresponding entries of the array representations. Using this representation, sample values are passed using array storage. Notable examples of this approach include the IEEE programs for digital signal processing (DSP Committee, 1979) and the Interactive Laboratory System (Signal

Technology, Inc).

Arrays have many problems as a signal representation. Arrays are finite in extent and can only be used to model discrete-time sequences. The representation is not immutable: the values of an array can be modified by any of the programs that reference it. In most languages, arrays are indexed from a fixed starting index, either 0 or 1, restricting the starting point of the non-zero support. Furthermore, indexing an array outside its support generally results in an error: that is, the domain of the representation is restricted to be identical to the non-zero support. Finally, in most languages, arrays do not have any provisions for maintaining associated properties, such as the cost of determining a sample value.

2.1.2 Stream-based signal representations

Stream-based signal representations are another common approach to signal representation, particularly in block-diagram programming languages (Kelly et al., 1961; Radar, 1965; Henke, 1975; Johnson, 1984). Within these languages, the user constructs a digital signal processing system by selecting and connecting a set of processing blocks, after instantiating their free parameters to specific values. The blocks represent signal-processing systems and the signals connecting them are generally represented by streams. In this usage, a stream is a data structure which behaves like a FIFO queue: values are read from the receiving end of the stream in the same order as they are placed into the stream on the transmitting end.[1] Depending on the stream implementation, attempts to retrieve sample values after the data in the stream has been exhausted can return either a unique "empty" symbol or a zero. Figure 2-1 illustrates the use of streams, assuming zeroes are transmitted once the data in the stream is exhausted.

Streams, while seemingly natural representations for one-dimensional discrete-time

[1]Streams, as they are described here, are different from the streams described in Abelson and Sussman (1985). As used here, reading a value from the stream has the side-effect of removing that value from the front of the stream. This corresponds more closely to queues as they are discussed in Abelson and Sussman (1985).

```
(DEFINE (IIR-STREAM A Y)             ; create y[n] = a^n u[n]
  (LET ((Y_N 1))                     ; the saved state of the system
    (PUT-STREAM Y
      (PROG1 Y_N                     ; use the saved state as the next value
        (SET! Y_N (* A Y_N))))))     ; update the state

(DEFINE (STREAM-ADD X1 X2 Y)        ; create y[n] = x1[n] + x2[n]
  (PUT-STREAM Y
    (+ (GET-STREAM X1) (GET-STREAM X2))))
```

Figure 2-1: Stream-based signal model

A stream is a data structure which behaves like a FIFO queue. In this example, GET-STREAM removes the next value from the receiving end of the stream. Values are removed in the same order as they are placed into the stream on the transmitting end by PUT-STREAM. Stream representations are shown for the IIR sequence class $y[n] = a^n u[n]$ and for the system, $y[n] = x1[n] + x2[n]$.

sequences, can not be used in a straightforward manner to represent multi-dimensional sequences, discrete-time Fourier-transform signals or z-transform signals. Furthermore, even within the field of one-dimensional discrete-time sequence representation, streams have some basic difficulties. Streams have an implied origin: the first value put into and taken out of the stream. This complicates the representation of left-sided sequences, like an anti-causal IIR sequence, and two-sided sequences, like a discrete-time sinc sequence. The non-zero support of the sequence is not explicitly available: to determine the support, all the sample values must be read from the stream and counted. Furthermore, the support may not be determinable at all: if the stream transmits zeroes after its data is exhausted, then in order to determine the non-zero support, an explicit function must be available which, when applied to the stream, indicates whether or not the stream is exhausted. Otherwise, there is ambiguity between a stream that has zero sample values followed by one or more non-zero sample values and a stream which was actually exhausted. Transmitting a unique "empty" signal to indicate an exhausted state avoids this problem at the expense of a nonuniform signal representation: in this case, the sample values outside the non-zero support on a stream will not be represented in the same way as those within the non-zero support. Finally, the FIFO behavior of the stream has the visible side-effects of queuing and dequeuing sample values and forces access to be sequential.

2.1.3 Object-based signal representations

Signal Representation Language (SRL)

SRL (Kopec, 1980; Kopec, 1985) is the result of research by Kopec into data abstractions both to reflect the basic characteristics of signals and to support numeric manipulations. In particular, Kopec (1980) advocated the immutability of signals and the explicit availability of their non-zero supports as being essential for simplifying and clarifying signal-processing programs. In SRL, sequences, being immutable, are not changed by subsequent processing; instead, new sequences are created. Thus, a sequence is represented as a distinct object whose sample values are provided on request. SRL also explicitly maintains a record of the size of the non-zero support of each sequence. Knowledge of the non-zero support of the sequence is used internally to simplify storage allocation, relieving the user of this chore. Two examples of sequence definition in SRL are shown in Figure 2-2.

As one of the first efforts in the field of signal representation, SRL (Kopec, 1985) was a welcome abstraction away from the typical representation of sequences as an array of values. Furthermore, by introducing the convention of immutability in sequences, signal-expression manipulation as it later developed was greatly simplified: the identity of a sequence was made context independent. The limitations of SRL in its applicability to numeric signal processing led to the later development of SPLICE, as described below. Among these limitations are the assumption that the non-zero support of a sequence always extends upward from the origin and the limitation that the sample values of a sequence can not be requested outside its non-zero support. Furthermore, since storage is simultaneously allocated for all the sample values in the non-zero support, sequences with an infinite non-zero support can not be represented.

```
(DEFSIGTYPE IIR-SIGNAL              ; the sequence class a^n(u[n] - u[n - M])
   :A-KIND-OF BASIC-SIGNAL          ; IIR-SIGNALs are also BASIC-SIGNALs
   :FINDER IIR                      ; function for creating an IIR-SIGNAL
   :PARAMETERS (A M)                ; parameters for describing an IIR-SIGNAL
   :INIT (SETQ-MY DIMENSIONS M)     ; the sequence length
   :FETCH ((INDEX)
          (IF (= INDEX 0) 1 (* A (SIGNAL-FETCH SELF (- INDEX 1))))))

(DEFSIGTYPE SUM-SIGNAL        ; the sequence class x1[n] + x2[n]
   :A-KIND-OF BASIC-SIGNAL    ; SUM-SIGNALs are also BASIC-SIGNALs
   :FINDER SIGNAL-ADD         ; function for creating a SUM-SIGNAL
   :PARAMETERS (X1 X2)        ; parameters for describing a SUM-SIGNAL
   :INIT (SETQ-MY DIMENSIONS ; the sequence length
                (MIN (SIGNAL-DIMENSIONS X1) (SIGNAL-DIMENSIONS X2)))
   :FETCH ((INDEX)
          (+ (SIGNAL-FETCH X1 INDEX) (SIGNAL-FETCH X2 INDEX))))
```

Figure 2-2: Signal representation in SRL

SRL represents signals using abstract data objects, with an identity distinct from the sample values. The length of the non-zero support is explicitly represented using the internal variable DIMENSIONS. The non-zero support is then assumed to cover the interval, from 0 to DIMENSIONS - 1. Represented here are the sequence classes $a^n(u[n] - u[n - M])$ and $x1[n] + x2[n]$

Signal Processing Language and Interactive Computer Environment (SPLICE)

SPLICE (Dove et al., 1984; Myers, 1986) was developed as a tool for numeric signal processing. It resulted from an effort to improve the computer representation of signals beyond the work that had already been done by Kopec (1980). In SPLICE, sequences continue to be immutable data objects, as they are in SRL. To allow for the representation of infinite length sequences, the computation of each sample value is delayed until it is explicitly needed. Like SRL, the non-zero support is explicitly represented but unlike SRL, no assumptions are made about its location: instead, a representation is provided for finite- and infinite-length intervals. Also, sample values outside the non-zero support can be accessed by the same operations that access the sample values inside the non-zero support: as expected, the values outside the support are returned as zero. Figure 2-3 illustrates this behavior. Furthermore, some basic facilities are provided for the

```
SPLICE: (HAMMING 255)                   ; create a centered, 255-point Hamming window
⟹ #<(HAMMING 255)>
SPLICE: (SUPPORT (HAMMING 255))         ; find the non-zero support interval
⟹ (INTERVAL -127 128)
SPLICE: (FETCH (HAMMING 255) 0)         ; get the sample value at n = 0
⟹ 1.0
SPLICE: (FETCH (HAMMING 255) -130)      ; get the sample value at n = −130
⟹ 0.0
SPLICE: (LISTARRAY (FETCH-INTERVAL (HAMMING 255) (INTERVAL -130 -123)))
                                  ; list the sample values for −130 ≤ n < −123
⟹ '(0.0 0.0 0.0 0.08 0.0801 0.0806 0.0813)
```

Figure 2-3: An example session in SPLICE
The token "SPLICE: " designates the user's input and the token "⟹ " designates the output.

maintenance of sequence properties, such as periodicity.

The SPLICE environment explicitly decouples certain issues and operations that are tied together in most software environments. Sequences, defined by the generating system and its inputs, behave uniformly independent of the signal-processing model used to define the system: for example, sample values of a sequence defined using a state-machine model can be fetched at any index without explicitly determining the previous states. This decouples the internal computational model from the calling convention for referencing the sample values. Since the creation of a sequence is separated from the computation of its sample values, sample values can be fetched at any index, from $-\infty$ to $+\infty$. Figure 2-4 provides examples of each of the computational models provided in SPLICE: the point-operator model which generates one sample value at a time; the array-operator model which generates multiple sample values simultaneously; the state-machine model which generates sample values sequentially using an internal state vector; and the composition model whose functionality is defined implicitly via the composition of other operators.

These and other generalizations were incorporated into SPLICE. This signal-processing package was integrated into the Lisp Machine environment and provides over 200 signal-processing operations within a common framework. The utility of SPLICE as a numeric

a. The point-operator model (generates one sample value at a time)

```
(DEFINE-SYSTEM SEQUENCE-ADD (X1 X2)
               (NUMERIC-SEQUENCE)      ; the output is a NUMERIC-SEQUENCE
  (SUPPORT ()
    (INTERVAL-COVER (SUPPORT X1) (SUPPORT X2)))
  (SAMPLE-VALUE (INDEX)       ; a definition for individual sample values within the support
    (+ (FETCH X1 INDEX) (FETCH X2 INDEX))))
```

b. The array-operator model (generates multiple sample values simultaneously)

```
(DEFINE-SYSTEM COMPLEX-FFT (INPUT N)
               (NUMERIC-SEQUENCE)      ; the output is a NUMERIC-SEQUENCE
  "The N-point FFT of INPUT"
  (SUPPORT () (INTERVAL 0 N))
  (COMPUTE-INTERVAL (DESIRED-INTERVAL)
    (INTERVAL 0 N))      ; the sample interval which should be computed simultaneously
  (INTERVAL-VALUES-COMPLEX (INTERVAL REAL-OUTPUT-ARRAY IMAG-OUTPUT-ARRAY)
      ; a definition for group of sample values within the support
    (ARRAY-COMPLEX-FFT
      (FETCH-INTERVAL INPUT (INTERVAL 0 N))
      (FETCH-IMAGINARY-INTERVAL INPUT (INTERVAL 0 N))
      REAL-OUTPUT-ARRAY IMAG-OUTPUT-ARRAY)))
```

c. The state-machine model (generates sample values sequentially using an internal state vector)

```
(DEFINE-SM-SYSTEM IIR-SEQUENCE (POLE-LOC)
                  (NUMERIC-SEQUENCE)      ; the output is a NUMERIC-SEQUENCE
  (STATE-MACHINE-START () 0)
  (INITIAL-STATE (STARTING-INDEX) 1)
  (CURRENT-VALUE (CURRENT-STATE INDEX)
      ; a definition for the sample values within the support of the state machine
    CURRENT-STATE)
  (NEXT-STATE (CURRENT-STATE INDEX)
    (* POLE-LOC CURRENT-STATE)))
```

d. The composition model (an implicit definition via the composition of other operators)

```
(DEFINE-COMPOSITION SINE-SEQUENCE (FREQUENCY)
  (SEQUENCE-SCALE 1/2     ; an implicit definition using the composition of other operations
    (SEQUENCE-ADD (COMPLEX-EXPONENTIAL-SEQUENCE FREQUENCY)
                  (COMPLEX-EXPONENTIAL-SEQUENCE (- FREQUENCY))))
  (ATOMIC-TYPE () :REAL))     ; information which is not available from the composition sequence
```

Figure 2-4: Signal representations in SPLICE

signal-processing environment is claimed to reduce program development times by factors of two to seven times (Myers, 1986).

2.2 Signal-Expression Manipulation Environments

This section reviews two pieces of work, reported by Myers (1986) and by Fogg (1988). Myers (1986) discusses an environment, E-SPLICE, which satisfies the criteria given earlier for a signal-expression manipulation environment. While the environment proposed by Fogg (1988) does not address many of these criteria, its focus on algorithm manipulation makes it relevant to the discussion of signal-expression manipulation environments.

2.2.1 Extended SPLICE (E-SPLICE)

E-SPLICE (Myers, 1986) provides the only previous environment with signal representations which meets all the criteria mentioned above: immutable objects, with the creation of signals being distinct from computation of their sample values; explicit signal properties, like non-zero support and bandwidth; distinct signal domain and non-zero support; and the ability to manipulate abstract signal objects. E-SPLICE (Myers, 1986) was built on the signal representation developed in SPLICE to allow for symbolic signal representation and manipulation. The main extensions provided by this addition are the ability to represent continuous-variable signals, like continuous-time signals or discrete-time Fourier transforms, and the ability to represent and manipulate abstract signals.

In representing discrete-time Fourier-transform signals, E-SPLICE opens an avenue to the manipulation of signals in the discrete-time Fourier-transform domain. Spectral characteristics, such as bandwidth and frequency support can be determined using information included in the system definitions.

E-SPLICE also supports the manipulation of what is referred to by Myers (1986) as abstract signals. Abstract signals are signals for which only a partial description has been given. The ability to represent and manipulate abstract signals effectively provides E-

SPLICE with the power to manipulate signal-processing systems instead of just signals, even though neither SPLICE nor E-SPLICE has an explicit representation for signal-processing systems. Abstract signals are used to represent a general signal description: conceptually, it provides the environment with the ability to refer to "some discrete-time sequence, $x[n]$" or "some discrete-time Fourier-transform signal, $X(e^{j\omega})$." By using these abstract signals as the inputs to the system of interest, the output characteristics of the system can be determined, without having to consider any compounding effects introduced by the identity of the input signal.

The power of signal-expression manipulations was demonstrated by the success of E-SPLICE in generating and ranking alternate implementations of a non-integer sampling rate conversion (Figure 2-5-a). Using the numbers of required additions and multiplications as a vector cost measure, a novel polyphase implementation of the non-integer sampling rate conversion was autonomously derived and determined to be efficient (Figure 2-5-b). This demonstration is particularly provocative, since soon after the completion of Myers' thesis, an independent article was presented on the subject of this new type of polyphase structure: not only did the example prove that automatic algorithm processing is viable but it generated a structure that was the subject of current research (Hsiao, 1987). This suggests that signal-expression manipulation may be useful not only for relieving much of the burden imposed by implementational details but also for the discovery of new, highly efficient implementations.

This work by Myers mapped out some of the possibilities in the area of signal-expression manipulation and demonstrated the viability of semi-automatic algorithm design. However, little effort was devoted to providing for the input of higher level knowledge, such as useful approaches to a problem. Nor is all of the available information exploited: the internal regularity of the system being manipulated was not used to guide the design process. Finally, various minor inconsistencies between abstract signals and completely specified signals remain.

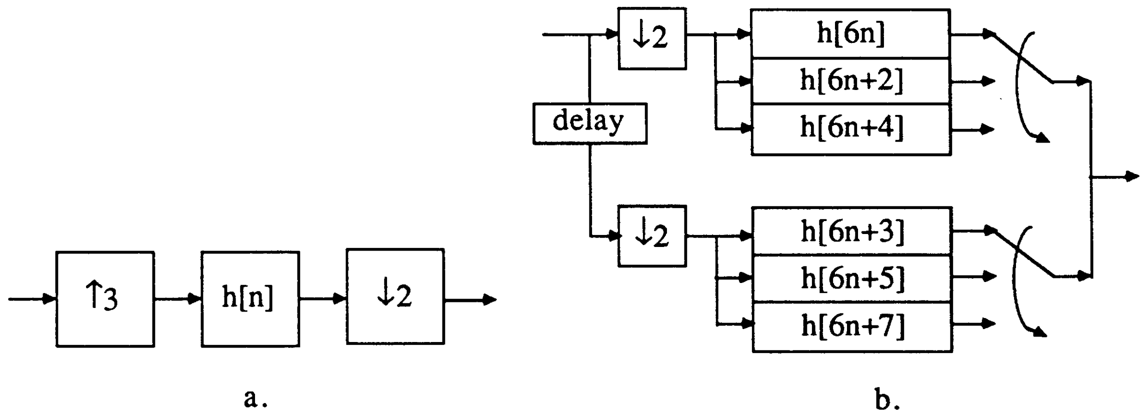


Figure 2-5: An example of signal manipulation in E-SPLICE
The power of signal-expression manipulation was demonstrated by E-SPLICE in its manipulation of the non-integer sampling rate conversion shown in part (a). In a search for efficient implementations, the polyphase structure shown in part (b) was autonomously derived. At the time E-SPLICE uncovered this implementation, this approach to non-integer sampling rate conversion was not present in classic multirate literature. An independent research effort has since presented this computational structure as a new, efficient method for non-integer sampling rate conversion.

2.2.2 Automatic Build-Up (ABU)

Fogg (1988) proposes to investigate the design of custom, irregular, signal-processing architectures. Building on the premise that every design must meet multiple performance criteria which can not be reliably reflected in a single functional value, ABU is a design environment aimed at exploring the tradeoffs between performance criteria or, more accurately, it is aimed at investigating approaches to this exploration. An example provided by Fogg (1988) of the proposed application area is the VLSI implementation of the Householder transform which transforms a general matrix into an upper triangular matrix: the competing performance criteria would be the area and throughput.

While the proposed application area of Fogg (1988), concentrating exclusively on low-level, highly irregular signal flow graphs, is dissimilar from the area considered in this thesis, his work is of interest for its proposed investigation of paradigms for controlling the search space. Among the techniques proposed are alteration strategies and decoupled design. Alteration strategies correspond to the equivalent-form transformations used in E-SPLICE (Myers, 1986) and in this thesis: they are a set of truth preserving rearrangements to the signal-processing algorithm. Decoupled design separates a single design problem into many subproblems which are treated independently. By exploring the subproblems independently, the size of the search space is reduced. This approach assumes that the subproblems can be solved without considering their interactions.

2.3 Summary

This chapter has developed an initial set of criteria for evaluating signal-processing representations and has examined some of the previous and concurrent work in the areas of numeric signal-processing environments and signal-expression manipulation environments using these criteria. Many of the characteristics that expedite numeric signal-processing were introduced by Kopec (1980): a distinct signal identity; an explicit non-zero support; and signal immutability. SPLICE (Dove et al., 1984; Myers, 1986) introduced the ability to manipulate signal properties and an explicit signal domain, distinct from the non-zero support. Characteristics that were considered essential to signal-expression manipulation include all these characteristics as well as the ability to generate and manipulate signal representations corresponding to a class or set of signals. Using this last set of capabilities to differentiate between numeric signal-processing environments and signal-expression manipulation environments, only E-SPLICE (Myers, 1986) can be categorized as a prior example of a signal-expression manipulation environment. The remainder of this thesis builds on the groundwork provided by Kopec (1980), Dove et al. (1984) and Myers (1986).

Chapter 3

Introduction of an Algorithm Design Environment: the design of an FSK-code detector

The stated goal of this thesis is to address the issues involved in providing a design environment. The desired environment would support the initial selection of a signal-processing algorithm, the symbolic and numeric description of the selected algorithm, and the manipulation of the algorithm to obtain alternate implementations. The power of such an algorithm design environment is illustrated in this chapter and throughout this thesis through the application of the Algorithm Design Environment (ADE) to the design of an FSK-code detector: this combination is used to provide concrete examples of the potential of design environments. These examples are presented at this point in the thesis in order to motivate the remaining discussion of the representational issues: the power of the concepts presented in later chapters will have already been demonstrated through the examples within this chapter. This organization has the added advantage that a casual reader can obtain an overview of the use of a signal-processing design environment, without being forced to consider its conceptual details.

3.1 Detection and Discrimination of FSK Codes for Multiple-beam Sonar

The problem of FSK-code detection and discrimination is used throughout this thesis to illustrate the issues under discussion. This section introduces the sonar application of FSK codes. A set of maximally separated codes are selected for transmission and a model for the reflected signal energy is introduced. Finally, a digital approximation to the optimal detectors for these signals is presented. It is this digital detection algorithm which is used within this thesis to illustrate the issues which arise in algorithm design.

Conventional sonar imaging systems achieve resolution either through the use of a single, swept beam or through the use of multi-element arrays. These techniques, while highly successful, present some inherent difficulties. In the case of the single swept beam, the time required to scan through the desired aperture can result in the failure to detect transients. When multi-element arrays are used instead, the hardware requirements necessary to achieve high resolution can result in a large, costly system. Jaffe and Richardson (Jaffe and Richardson, 1989; Richardson, 1989) propose an alternative to these two techniques using the simultaneous transmission of a set of coded waveforms.

The transmitter in the proposed system is a set of N transducers, each illuminating a different direction and each transmitting a distinct signal, S_i for $i = 1, ..., N$. One wide-beam hydrophone is used as a receiver. Multiple-hypothesis testing is then used to detect and discriminate the returns from the separate beams. In order to achieve good spatial resolution, the set of signals $\{S_1, ..., S_N\}$ must have good signal-to-signal rejection for all possible time delays. In addition, to achieve good range resolution, each signal should have a sharply peaked autocorrelation function. Jaffe and Richardson (1989) present a mathematical derivation for a set of FSK codes with these properties:

$$\begin{aligned} S_i(t) &= \sum_{k=1}^{N} Re\{P_{i,k}(t)\} \quad \text{for } i = 1...N \\ P_{i,k}(t) &= C_l(t - kT) \end{aligned}$$

where $l = p(i,k)$ provides a different permutation of the numbers $1, ..., N$ versus k for each i

$$C_l(t) = w(t)e^{j2\pi(f_c + \frac{M(l-1)}{T})t}$$

where $w(t) = 0$ for $t < 0$ and for $t \geq T$

From the way $P_{i,k}(t)$ and $C_l(t)$ are defined, each signal is made up of a sum of N individual, uniformly-spaced frequency bursts (commonly referred to as frequency chips). When $N+1$ is prime, $f(i,k)$ can be selected such that the signals and all their circular shifts achieve maximal Hamming distance separation.[1] The window $w(t)$ allows the frequency chips to be shaped to adjust their side-lobe characteristics.

The received signal can be modeled as a superposition of the reflected energy from each of the illuminated scattering centers:

$$r(t) = \sum_m \sum_{i=1}^{N} \sum_{k=1}^{N} \rho_{m,i} Re\{e^{j\varphi_{i,k}} P_{i,k}(t - \tau_m)\}$$

The summation over m represents the superposition of the returns from multiple scattering centers. $\rho_{m,i}$ is used to represent the strength of the return from the m'th scattering center: if the scattering center lies in the i'th beam, $\rho_{m,i}$ is a function of the scattering cross-section of the target as well as its range; otherwise, $\rho_{m,i} = 0$. τ_m is the propagation delay for the combined forward and return paths. $\varphi_{i,k}$ represents a nonuniform phase distortion introduced by the scattering characteristics of the object and the fluctuations in the propagating medium. The possibility of a Doppler frequency shift is ignored in this model.

From this model, the parameters $\rho_{m,i}$, $\varphi_{i,k}$ and τ_m are unknown parameters and must be determined to invert the imaging process. $\rho_{m,i}$, once corrected for the variations due to range, can be used as a gray-scale representation of the cross-section of an object, and thus is not considered further. Using this model, the discrete-time approximation to the optimal detectors for the N signal beams is shown in Figure 3-1. This algorithm uses matched filters to detect the individual frequency chips. Since the model allows for an

[1]The Hamming separation distance is the number of elements that differ between any two code words.

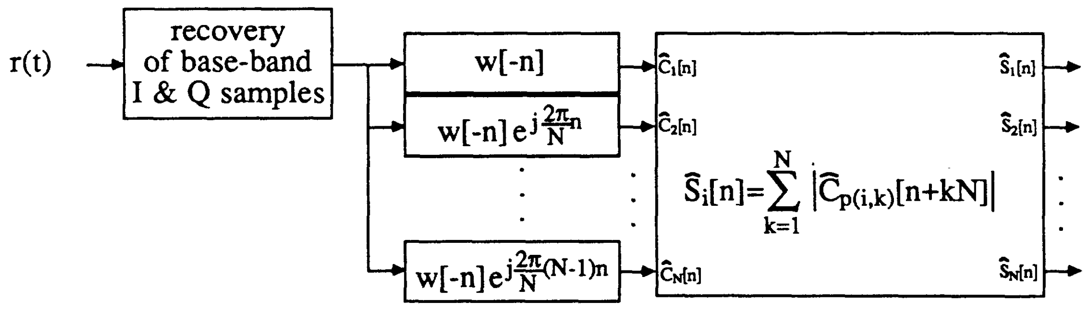

r(t)
recovery of base-band I & Q samples
w[-n]
$w[-n]e^{j\frac{2\pi}{N}n}$
$w[-n]e^{j\frac{2\pi}{N}(N-1)n}$
$\hat{C}_1[n]$
$\hat{C}_2[n]$
$\hat{C}_N[n]$
$\hat{S}_i[n]=\sum_{k=1}^{N}|\hat{C}_{p(i,k)}[n+kN]|$
$\hat{S}_1[n]$
$\hat{S}_2[n]$
$\hat{S}_N[n]$

Figure 3-1: The discrete-time approximation to the optimal detectors for N FSK-coded sonar signal beams

The structure shown here implements the discrete-time approximation to the optimal detector assuming an unknown phase history and an unknown time delay. Matched filters are used to detect the individual frequency chips. Since there can be a nonuniform phase distortion between frequency chips, the outputs from these subfilters are combined incoherently.

unknown, nonuniform phase distortion between frequency chips, incoherent summation must be used across the matched frequency chips: this incoherent combination is completed in the last box in Figure 3-1. Finally, since there is an unknown time delay in the return, the output from these detectors is desired at each point in time.

3.2 Algorithm Design Environment (ADE)

The stated goal of this thesis is to explore the issues involved in providing a design environment supporting the initial selection of a signal-processing algorithm, the symbolic and numeric description of the selected algorithm, and the manipulation of the algorithm to uncover alternate implementations. The Algorithm Design Environment (ADE) is an environment which has been implemented based on the ideas presented in this thesis and which will be used to demonstrate these ideas. This section provides a brief description of ADE. This description is guided by a discussion of two short sessions in ADE, one illustrating the programming of the environment and the other, its interactive use.

ADE is a descendant of the SPLICE and E-SPLICE environments, described in Chapter 2. ADE inherits its basic approach to signal definition and representation from SPLICE (Dove et al., 1984; Myers, 1986). The influences of E-SPLICE (Myers, 1986) and to a lesser extent PDA (Dove, 1986) are reflected in the structure of some parts of the rule base. In particular, as in E-SPLICE, ADE uses backward-chaining rules to describe the properties of signals. ADE, like E-SPLICE, supports multilevel matching within the patterns of these rules. The approach used in ADE for matching forward-chaining rules was introduced by Dove (1986). ADE also makes use of a subset of QM (Sacks, 1982) and a limited number of functions from MACSYMA (Mathlab Group, 1983). QM (Sacks, 1982) is the product of research into qualitative mathematics. It represents, manipulates and describes piecewise-continuous functions. A subset of QM is used to record and propagate constraints on symbolic numbers. ADE includes an extension to QM to support limited reasoning about symbolic integers as well as the continuously variable numbers. ADE also makes limited use of MACSYMA (Mathlab Group, 1983) to simplify and factor the polynomials used in the characterization of z-transform signals. ADE is written in Symbolics Common Lisp (Symbolics, 1986). This choice of language provides both the flexibility of a LISP dialect and support for object-oriented programming.

The remainder of this section provides examples of the use of ADE in the context of the FSK-code problem introduced above. Examples are given of both programming and using the environment.

In the sonar imaging problem, the actual problem is to find a way to achieve good spatial resolution in a sonar system without paying for this resolution by either of the two traditional penalties. The first step in solving this problem is to select a method by which it will be solved. The majority of this selection is a matter of signal-processing experience and creativity. However, this selection process can be accelerated by providing a support environment in which the signal and system representations closely match the internal models used by the system designer. These representations must change according to the problem at hand, since different problems give rise to different signal models. To provide this adaptation of representations, ADE allows the system designer to introduce his own

```
1  (DEFINE-SYSTEM-CLASS-ALIAS
2            (INCOHERENT-COMBINATION N-CODES@INTEGER PERMUTATION@FUNCTION)
3                ; accept an integer and a function as system parameters
4            (FILTER-OUTPUTS@2D-SEQUENCE)       ; accept a 2D-SEQUENCE as an input
5        (SHIFT-INVARIANT-SYSTEM HOMOGENEOUS-SYSTEM 2D-SYSTEM)
6            ; a subclass of these classes
7    ("the system which incoherently combines shifted versions of the sequences in FILTER-OUTPUTS,
8       using PERMUTATION to determine the order in which they are combined"
9    SELF)      ; the systems "alias" only themselves
10   NIL ()       ; generate a new output signal class, without any additional superior signal classes
11   ("the output sequences from the incoherent combination"
12       ; the output signals "alias" the composition of operations:
13       ; (BANK-OF-SEQUENCES S_1 ... S_N)
14       ; where S_i = (SEQUENCE-ADD P_{i,1} ... P_{i,N})
15       ; where P_{i,k} = (OUTPUT-OF (SHIFT (* k N)) (MAGNITUDE C_{p(N,i,k)}))
16      (MAP-OVER 'BANK-OF-SEQUENCES I 1 (1+ N-CODES)
17        (MAP-OVER 'SEQUENCE-ADD K 1 (1+ N-CODES)
18          (OUTPUT-OF (SEQUENCE-SHIFT (* K N-CODES))
19            (SEQUENCE-MAGNITUDE
20              (FETCH-SEQUENCE FILTER-OUTPUTS
21                                 (- (PERMUTATION N-CODES I K) 1))))))))
```

Figure 3-2: An example of the programming of ADE

The new signal or system classes be added to ADE, as illustrated here. The lines of this definition are numbered to the left, for ease of reference.

signal and system definitions. For example, in the FSK-code detector shown in Figure 3-1, the incoherent combination of the matched filter outputs is modeled as a single processing block which follows but is separate from the matched filtering itself. To support this model of the detector, Figure 3-2 defines a new system class, INCOHERENT-COMBINATION. This definition also creates a new signal class INCOHERENT-COMBINATION-OUTPUT, which contains the output signals from INCOHERENT-COMBINATION systems. The definition relies on the composition of other, previously defined signal-processing systems to provide the output signals with their observable characteristics: lines 16–21 of Figure 3-2 describe this composition.

To simplify the programming task, signal and system definitions closely mimic the notational conventions used in signal processing. As illustrated in Figure 3-2, signal and system definitions form new "classes" of signals and systems. Hierarchies of classes are

```
1  (DEFINE-SIGNAL-CLASS
2      COMPLEX-EXPONENTIAL-SEQUENCE (FREQUENCY@REAL-NUMBER)
3          ; accept one real number as a parameter
4        (COMPLEX-EXPONENTIAL DISCRETE-TIME-SEQUENCE)      ; a subclass of these classes
5    :CANONICALIZE-PARAMETERS (SETQ FREQUENCY ($- ($MOD ($+ FREQUENCY PI) 2PI) PI))
6        ; only use −π ≤ FREQUENCY < π
7    :NON-ZERO-SUPPORT [MINF INF]      ; a doubly infinite non-zero support
8    :FT NONE      ; no Fourier transform: avoids DTFT impulses
9    :ZT NONE      ; no z transform
10   (GOAL PERIODICITY      ; the periodicity of dc is one sample
11                          ; otherwise, it is the smallest integer which is a multiple of the basic periodicity
12     :NAME COMPLEX-EXPONENTIAL-PERIODICITY
13     :OBJECT ?SELF
14     :ANSWER (IF ($= FREQUENCY 0)
15                 1 (DISCRETE-PERIODICITY ($ABS ($/ 2PI FREQUENCY))))
16     :DONE)      ; explicitly terminate search
17   (GOAL SIMPLIFICATION      ; simplify dc as a constant
18     :NAME ZERO-FREQUENCY-AS-CONSTANT
19     :OBJECT (SPECIFIC-MEMBER COMPLEX-EXPONENTIAL-SEQUENCE 0)
20     :ANSWER (CONSTANT-SEQUENCE 1)))
```

Figure 3-3: An example of an inherent signal class definition

Signal classes can be defined independently as inherent signal classes. The definition of the inherent signal class COMPLEX-EXPONENTIAL-SEQUENCE is shown here. The lines of this definition are numbered to the left, for ease of reference.

used to make similarities explicit and to reduce the amount of coding required. Signals are formed by one of two paths: either as independent entities which are inherently defined, like an impulse or a complex-exponential sequence, or as the output from a system which has been applied to some inputs. An example of an inherent signal-class definition is shown in Figure 3-3. Some of the 43 inherent signal classes currently defined in ADE are listed in Table 3.1. Defining forms are also provided to allow for additional signal-class definitions (see Appendix A).

In contrast with the definitions of inherent signal classes, the definition and characterization of system output signals is actually part of the definition of the system class and, as such, is syntactically tied to the system-class definition. An example of a system-class definition was provided in Figure 3-2. Some of the 169 system classes currently defined

Table 3.1: Some of the system and inherent signal classes currently defined in ADE

Inherent signal classes (43 hierarchical classes)

DISCRETE-TIME-SEQUENCE	COMPLEX-EXPONENTIAL
FOURIER-DOMAIN-SIGNAL	CAUSAL-RECTANGULAR-WINDOW
Z-DOMAIN-SIGNAL	SINC
2D-SEQUENCE	COSINE-SEQUENCE
RATIONAL-ZT	SINE-SEQUENCE
CONSTANT	FIR-SEQUENCE
POWER-SEQUENCE	IIR-SEQUENCE
IMPULSE	CAUSAL-IIR-SEQUENCE
GENERAL-EXPONENTIAL	ANTICAUSAL-IIR-SEQUENCE
UNIT-STEP-SEQUENCE	STABLE-IIR-SEQUENCE
CAUSAL-HAMMING-WINDOW-SEQUENCE	⋮

System classes (169 hierarchical classes)

DISCRETE-TIME-SYSTEM	ADD	BANK-OF-SEQUENCES
2D-SYSTEM	SUBTRACT	ROTATED-BANK-OF-SEQUENCES
FOURIER-DOMAIN-SYSTEM	MULTIPLY	SHORT-TIME-WINDOW
Z-DOMAIN-SYSTEM	CONVOLVE	SHORT-TIME-FT
SHIFT-INVARIANT-SYSTEM	SHIFT	MAPOVER-SYSTEM
GENERALIZED-SHIFT-INVARIANT-SYSTEM	SCALE	FIR-FILTER
MEMORYLESS-SYSTEM	RECIPROCAL	CAUSAL-IIR-FILTER
ASSOCIATIVE-SYSTEM	DIVIDE	ANTICAUSAL-IIR-FILTER
ADDITIVE-SYSTEM	REAL-PART	SIGNAL-ALIAS-IN-2PI
HOMOGENEOUS-SYSTEM	IMAG-PART	FOURIER-TRANSFORM
GENERALIZED-HOMOGENEOUS-SYSTEM	MAGNITUDE	INVERSE-FOURIER-TRANSFORM
LINEAR-SYSTEM	INPUT-PHASE	Z-TRANSFORM
GENERALIZED-LINEAR-SYSTEM	ABSOLUTE-VALUE	INVERSE-Z-TRANSFORM
SEQUENCE-CIRCULAR-SHIFT	SCALE-INDEX	INVERSE-TRANSFORM
SEQUENCE-CIRCULAR-REVERSE	UPSAMPLE	DISCRETE-FOURIER-TRANSFORM
SEQUENCE-CIRCULAR-CONVOLVE	DOWNSAMPLE	COMPLEX-CONJUGATE
SEQUENCE-CONVOLVE-OVERLAP-SAVE	INTERLEAVE	⋮

in ADE are listed in Table 3.1. Defining forms are again provided to allow for additional system-class definitions.

Once all the appropriate signal and system classes have been defined, the process of creating and analyzing the signals and systems involved in the design problem is greatly simplified. Using the previous definition, the FSK-code detector is easily described, as shown on line *I-4* of Figure 3-4. Furthermore, as can be seen by comparing the input on line *I-4* with the model for the detector shown in Figure 3-1, the computer representation and the designer's representation are closely matched. As is shown in the remainder of this figure, ADE provides information about the properties of the output signals from this detector.

In more detail, line *I-1* of Figure 3-4 defines a function for generating the permutations of the FSK-code frequency chips. Line *I-2* provides a partial description of a window which will be used to shape the frequency chips: the window is a real-valued, discrete-time sequence whose non-zero support extends from 0 to 15 (i.e. $16 - 1$) and whose values range between 0 and 1. This description is only a partial description of the window since there are a large number of discrete-time sequences which satisfy all parts of this description. The resulting object, printed on line *O-2*, is an abstract signal.

Line *I-2* uses intervals to characterize the window sequence. Intervals are used in ADE to describe sets of numbers. The examples of their use in line *I-2* are [0 16], describing the non-zero support of the discrete-time sequence; {0 1}, describing the range for the real part of the sample values of the sequence; and {0 0}, describing the range of the imaginary part of the sample values. These examples include two distinct types of intervals: the non-zero support of the discrete-time sequence is an interval containing only integers while the other two intervals contain all the numbers lying between their end points. The discrete interval, [*start* *end*], represents the set of integers, n, such that $start \leq n < end$. In discrete intervals, the starting and ending points must be either integers or real numbers. If no integers lie in the interval, then a unique empty interval is returned. For example, both [-0.5 π] and [0 4] will return the same interval

```
I-1 ADE: (DEFUN MOD-N+1 (N I K)
           "provide different permutations of the numbers 1, ..., N versus k for each i"
           (IF (PRIMEP (1+ N))
               (MOD (* I K) (1+ N))
               (ERROR "Can not get maximal separation unless N+1 is prime")))
O-1 ⟹ MOD-N+1
I-2 ADE: (NAMED-SETQ
           WINDOW (REVERSE (A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE &PROPERTIES
                                   :NON-ZERO-SUPPORT [0 16]
                                   :SAMPLE-TYPE 'EXTENDED-REAL-NUMBER
                                   :RANGE (CREATE-RANGE {0 1} {0 0}))))
O-2 ⟹ #<WINDOW>
I-3 ADE: (NAMED-SETQ MAX (A-MEMBER-OF 'REAL-NUMBER &PROPERTIES :> 0))
O-3 ⟹ #<MAX>
I-4 ADE: (NAMED-SETQ
           DETECTOR-OUTPUT
           (OUTPUT-OF (INCOHERENT-COMBINATION 16 'MOD-N+1)
             (OUTPUT-OF (MODULATED-FILTER-BANK WINDOW 16)
               (A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE &PROPERTIES
                 :RANGE (CREATE-RANGE {($- MAX) MAX} {($- MAX) MAX})))))
O-4 ⟹ #<FSK-DETECTOR-OUTPUT>
I-5 ADE: (RANGE FSK-DETECTOR-OUTPUT)
         ; determine the range of (BANK-OF-SEQUENCES $S_1$ ... $S_N$)
         ;          where $S_i$ = (SEQUENCE-ADD $P_{I,1}$ ... $P_{I,N}$)
         ;          where $P_{i,k}$ = (OUTPUT-OF (SHIFT (* k N)) (MAGNITUDE $C_{F(N,i,k)-1}$))
         ;          where $C_l$ = X1 * ($e^{-j\frac{2\pi}{N}ln}$ WINDOW)
         ;          where X1 is the abstract input sequence
         ;                         ⋮
         ; determine the range of X1 * ($e^{j0}$ WINDOW)
         ;                         ⋮
         ; determine the range of X1 * ($e^{-j\frac{2\pi}{N}n}$ WINDOW)
         ;                         ⋮
O-5 ⟹ #<(RANGE {0 ($* 32 MAX)} {0 0})>
I-6 ADE: (NAMED-SETQ RECTANGULAR
                   (REVERSE (CAUSAL-RECTANGULAR-WINDOW-SEQUENCE 16)))
O-6 ⟹ #<RECTANGULAR>
```

Figure 3-4: A sample of an interactive session in ADE

The *I*-lines with the token "ADE: " designate the user's inputs and the *O*-lines with the token "⟹" designate the outputs. See the text for a detailed discussion.

```
I-7 ADE: (PERIODICITY (OUTPUT-OF (INCOHERENT-COMBINATION 16 'MOD-N+1)
                       (OUTPUT-OF (MODULATED-FILTER-BANK RECTANGULAR 16)
                         (A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE &PROPERTIES
                           :PERIODICITY 256))))
         ; determine the periodicity of (BANK-OF-SEQUENCES $S_1$ ... $S_N$)
         ;          where $S_i$ = (SEQUENCE-ADD $P_{I,1}$ ... $P_{I,N}$)
         ;          where $P_{i,k}$ = (OUTPUT-OF (SHIFT (* k N)) (MAGNITUDE $C_{F(N,i,k)-1}$))
         ;          where $C_l$ = X2 * ($e^{-j\frac{2\pi}{N}ln}$ RECTANGULAR)
         ;          where X2 is the abstract input sequence
         ;                    :
         ; determine the periodicity of X2 * ($e^{j0}$ RECTANGULAR)
         ;                    :
         ; determine the periodicity of X2 * ($e^{-j\frac{2\pi}{N}n}$ RECTANGULAR)
         ;                    :
O-7 ==> 256
I-8 ADE: (PERIODICITY (OUTPUT-OF (MODULATED-FILTER-BANK RECTANGULAR 16)
                       (A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE &PROPERTIES
                         :PERIODICITY 256)))
O-8 ==> 256
```

Figure 3-4 continued.

containing the integers 0, 1, 2 and 3 and both [π 4] and [0 -4] will return the empty interval. The continuous interval $\{start \;\; end\}$ represents the set of numbers, z, such that $Im(z) = Im(start) = Im(end)$ and $Re(start) \leq z < Re(end)$. From this description of continuous intervals, the starting and ending points can be complex numbers, as long as their imaginary parts are equal. If $Re(start) > Re(end)$, then no numbers will lie in the interval and the unique empty interval is returned. Finally, the interval $\{point \;\; point\}$ is used to represent the continuous interval containing only the single number, *point*. As shown in Appendix A, a wide variety of interval manipulation functions are provided in ADE.

Line *I-3* of Figure 3-4 creates a symbolic, positive, real-valued number. As will be illustrated in line *I-4*, symbolic numbers can be used to describe abstract signals. Constraints can be imposed both on the type of the symbolic number and on the relative or absolute magnitude of the symbolic number: a type constraint insures that the symbolic number of line *O-3* is real-valued as opposed to complex-valued and a magnitude constraint forces its value to be greater than zero.

Line *I-4* of Figure 3-4 creates an incoherent detector for the set of 16 FSK codes which uses MOD-N+1 to determine the frequency-chip permutations and which uses WINDOW to shape each frequency chip. The input to this detector is an abstract complex-valued, discrete-time sequence whose real and imaginary ranges are bounded by the symbolic numbers ±MAX. Line *I-5* of Figure 3-4 requests the range of the output from this application. The range of the signal is one of its properties and, as illustrated on line *I-5* can be explicitly requested. Signal properties, such as symmetry, sample type and non-zero support, are explicitly available characteristics of every signal. Similarly, system properties, such as equivalent systems and invertibility, are explicitly available characteristics of every system. Some of the signal and system properties which are currently included in ADE are listed in Table 3.2.

Some of the intermediate range computations which ADE makes in determining the range of the FSK-code detector output are noted between the input line *I-5* and the output line *O-5*. To determine the result shown on line *O-5*, a variety of extended algebraic and trigonometric functions are used to manipulate the symbolic numbers which describe the range. The majority of these extended algebraic and trigonometric functions produce a symbolic number as their output given a symbolic input.

Lines *I-7* and *I-8* request the periodicity of the output from the incoherent detector and the periodicity of the output from the matched filter bank, respectively. The input signals to both of these operations are abstract periodic, discrete-time sequences. This characterization of the digitized FSK-code returns is an appropriate model for a static field of scattering centers. Some of the intermediate periodicity determinations which ADE makes in arriving at line *O-7* are noted between the input line *I-7* and the output line *O-7*. None are noted between the input line *I-8* and the output line *O-8*, since no intermediate periodicity determinations are made. This immediate response to line *I-8*

Table 3.2: Some of the signal and system properties currently supported by ADE

INVERTIBLE-P:	T or NIL.
INVERSE-SYSTEM:	a system or #<UNKNOWN>.
SAMPLE-TYPE:	a data type.
RANGE:	a range of values described by the ranges of the real and imaginary parts.
NON-ZERO-SUPPORT:	any interval.
PERIODICITY:	any non-negative number.
SYMMETRY:	any symmetry descriptor.
COMPUTABLE-P:	T or NIL.
SAMPLES-COMPUTABLE-P:	T or NIL.
FT:	any discrete-time Fourier-transform signal.
IFT:	any discrete-time sequence.
ZT:	any z-transform signal.
IZT:	any discrete-time sequence.
ROC:	any interval of radii covered by $\{0\ \infty\}$, the null interval or #<UNKNOWN>.
POLES:	any polynomial or #<UNKNOWN>.
ZEROES:	any polynomial or #<UNKNOWN>.
COST:	any cost descriptor.
EQUIVALENT-FORMS:	a list of equivalent signals or systems.
EFFICIENT-IMPLEMENTATIONS:	a list of equivalent signals or systems which are computationally efficient.
SIMPLIFICATION:	the original or a simpler signal or system.

is due to information sharing between related abstract objects, as will be discussed in Chapter 4.

This section has attempted to illustrate some of the manipulations supported by the Algorithm Design Environment. In addition to allowing the manipulation of simple properties, like range and periodicity, the environment supports the autonomous search for alternate descriptions of a signal-processing expression and for computationally efficient implementations. Additional examples from ADE will be introduced later in this chapter.

3.3 Derivation and Ranking of Equivalent Algorithms

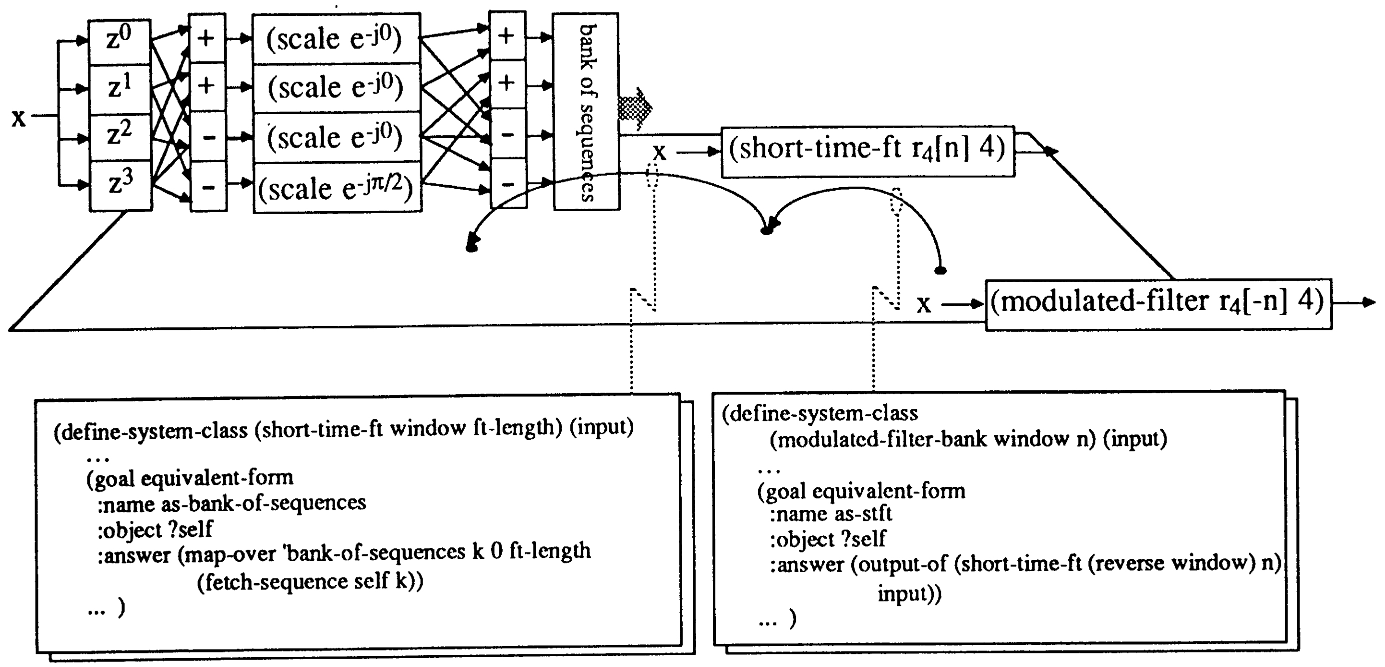
There are many examples in which reconfiguring an algorithm through a large number of straightforward transformations can lead to major gains in performance. For example, the computation of the Fourier transform is very rich in the variety of transformations that can be applied to reconfigure the algorithm and, as the FFT clearly demonstrates, the resulting efficiency can differ by orders of magnitude. In order to find the applicable transformations and to explore the full design space, the environment must search through the space of algorithms which are input/output equivalent to a given signal-processing expression. This section describes the search for equivalent implementations and then introduces constraints to limit that space.

3.3.1 Unconstrained search for equivalent algorithms

The task of finding alternate descriptions or implementations of a signal-processing expression is the same as finding all the identity transformations which are applicable to the signal-processing expression or one of its subexpressions. For example, to find the equivalent implementations of the filter bank shown in Figure 3-1, all the applicable identity transformations for the filter bank should be completed as should the transformations on the modulated window sequences and the input sequence. In addition, once an alternate description is uncovered, all of the identity transformations which are applicable to this new description or to one of its subexpressions must also be applied. Thus, equivalent implementations of a signal-processing expression can be obtained in any of a variety of ways: a transformation can be applied to the original signal-processing expression itself; a subexpression of the original expression can be replaced by an equivalent implementation of the subexpression; or either of these approaches can be applied to one of the newly uncovered equivalent implementations of the signal-processing expression.

To simplify this discussion, a graphical representation of the search process is presented in Figure 3-5. Using this representation, the problem of finding the equivalent forms of a signal-processing expression, without consideration of its subexpressions, is represented graphically as a net, as shown in Figure 3-5-a. One of the nodes of this net represents the starting signal-processing expression. The remaining nodes of the net represent equivalent implementations of the original expression. Some of these nodes are connected directly to the original node via simple transformation rules. These newly obtained nodes can themselves be used as seeds for other transformations: this recursive search process is encoded by a control strategy RECURSIVE-EQUIVALENT-FORMS, which will be shown in Figure 5-1-a. This search for additional nodes stops when no new nodes remain to be considered.

Any of the nodes of this net can also be viewed as a combination of subexpressions. The subexpressions of a generated signal are the generating system and its inputs and the subexpressions of a system or an inherent signal are the parameters of the class. Each of these subexpressions can also be manipulated. In particular, if the subexpression is itself a signal or system, its equivalent implementations can be used to replace it in an enclosing expression. Graphically, requesting the equivalent forms of a subexpression drops the problem down to another net and again tries to find connected nodes (Figure 3-5-b). The set of nodes found on this lower net is then projected back up into the original net by replacing the subexpression in the enclosing expression with its equivalent forms, as shown in Figure 3-5-c. This projection can generate new nodes in the original net. The decomposition and recomposition process is encoded in the control strategy



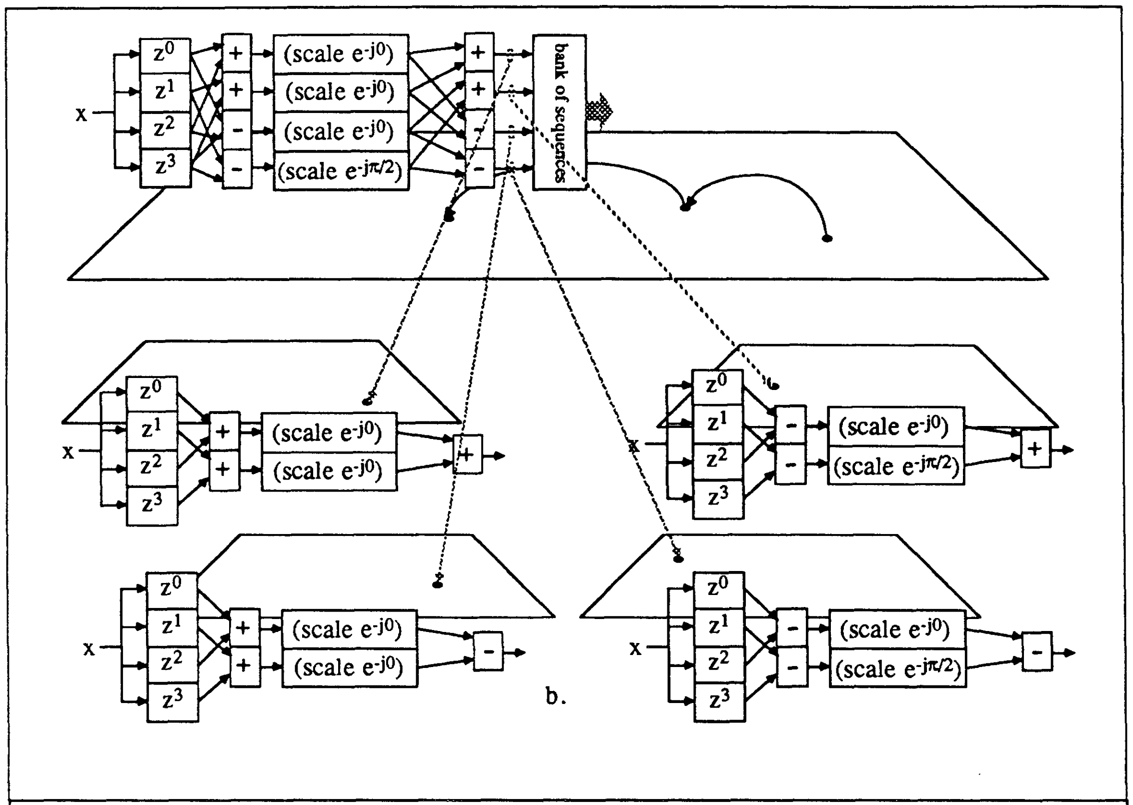
```
(define-system-class (short-time-ft window ft-length) (input)
   ...
   (goal equivalent-form
    :name as-bank-of-sequences
    :object ?self
    :answer (map-over 'bank-of-sequences k 0 ft-length
                (fetch-sequence self k))
   ... )
```

```
(define-system-class
   (modulated-filter-bank window n) (input)
   ...
   (goal equivalent-form
    :name as-stft
    :object ?self
    :answer (output-of (short-time-ft (reverse window) n)
                input))
   ... )
```

a.

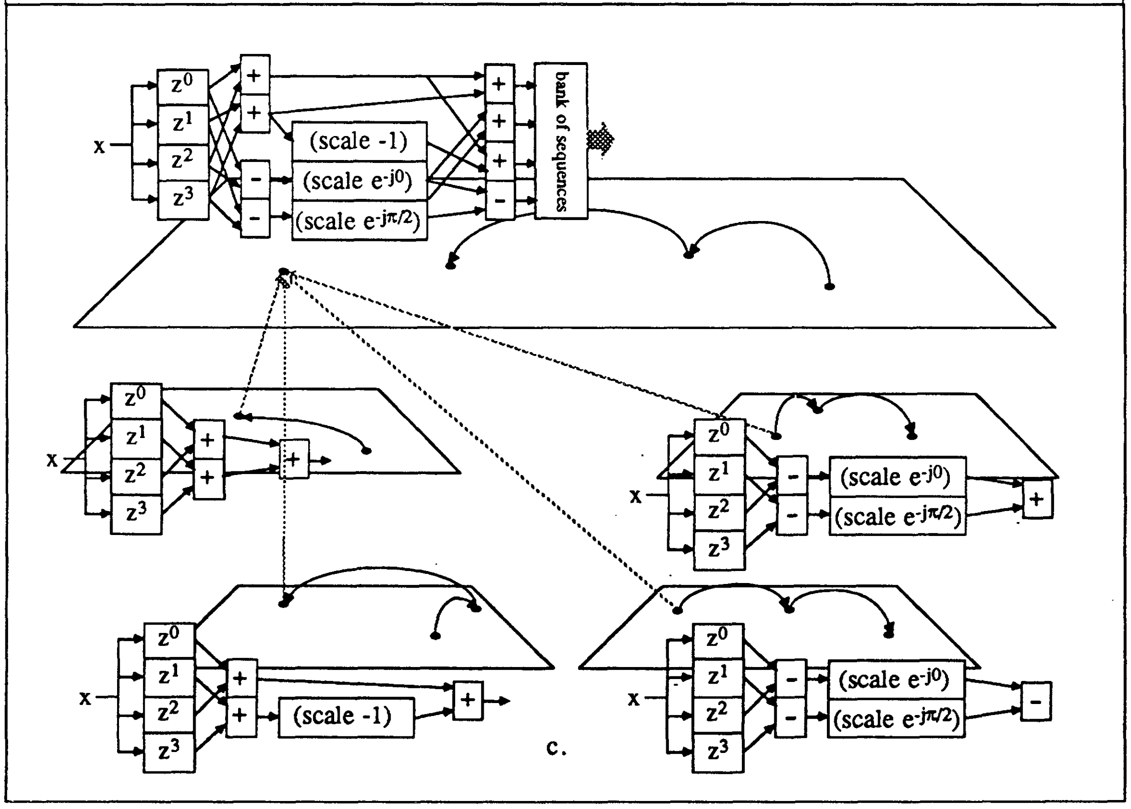
Figure 3-5: A net representation of the search for equivalent forms

(a) A net representation of the process of finding equivalent forms is shown here. The problem of finding the equivalent forms of an object, without manipulating its generating components, can be represented by the problem of tracing out a connected net: each of the nodes in the net is an equivalent object and the simple transformation of one object into an equivalent object is represented by a directed arc labeled by the transformation rule.
(b) Assuming that the objects under consideration are signals or systems, each of the nodes of this net can be viewed as a composite object and each of their components can be manipulated. In particular, the equivalent forms of any of the components can be used to replace that component in the generating form of the node. This replacement generates a new node with the same equivalence relationships as the manipulated node. Requesting the equivalent forms of an input or parameter graphically drops the problem down to another net and again tries to find related points.
(c) The set of points found on this lower net is then projected back up into the original net by using each of these points as inputs to the removed generating function.



x
z0
z1
z2
z3
+
-
(scale e-j0)
(scale e-jπ/2)
bank of sequences

b.



x
z0
z1
z2
z3
+
-
(scale -1)
(scale e-j0)
(scale e-jπ/2)
bank of sequences

c.

EQUIVALENT-FORMS-BY-PARTS, which will be shown in Figure 5-1-b. These new nodes are also used as seeds for finding additional nodes through recursive transformation and through expression decomposition.

The same strategies for finding the equivalent implementations of an expression are obviously also applicable to finding the equivalent implementations of any of the subexpressions. Thus, each of the searches for the equivalent forms of the subexpressions can also give rise to subsearches, using some even lower net. The downward progression stops when there are no more subexpressions which are signals or systems.

3.3.2 Constraints to avoid combinatorial growth of the algorithm design space

As described above, the search for equivalent implementations of a signal-processing expression must consider the equivalent implementations of the subexpressions as well as the complete expression itself. Since each of the subexpressions are independently manipulated and independently recombined to form new equivalent expressions, the size of the search space under consideration grows combinatorially with the number of subexpressions. To illustrate, consider the problem of implementing the full FSK-code detector for sixteen channels. Five independent descriptions of a simple, finite-length convolution are embedded in ADE: the direct-form convolution; the overlap-save convolution; the Fourier-domain representation of convolution; the z-domain representation of convolution; and the representation of convolution as the sum of scaled, shifted versions of the input. Thus, using these subexpressions as inputs into the incoherent summation, there will be $5^{16} \approx 10^{11}$ equivalent forms to consider.[2] Each of these implementations would then be reconsidered to see if any additional equivalent forms could be found, due to interactions between the implementations of the matched filters and the implementations of the incoherent processing.

[2]None of these implementations exploit the special structure of the modulated filter bank. The actual number of equivalent implementations which have to be considered is more than 10^{19}.

The approach to limiting the search space which is advocated in this thesis exploits the internal regularity of signal-processing algorithms. Signal processing algorithms are often described at different levels of detail: for example, the incoherent addition of a two-dimensional input sequence can be described by #<(OUTPUT-OF (INCOHERENT-COMBINATION 4 MOD-N+1) X)> or by the structure shown in Figure 3-6. From the high-level description of the algorithm, the regularity in the low-level computational structure can often be asserted: from the high-level description given by the INCOHERENT-COMBINATION system, the underlying regularity inherent in Figure 3-6 can be asserted. By enforcing these internal correspondences in the low-level descriptions, the space of equivalent forms which is explored can be drastically reduced. This approach to pruning the search is heuristic. However, the regularity of the computation suggests that the efficient implementations will reflect the same regularity: if separate sections of an algorithm are very similar, then the efficient implementations of these separate sections are likely to coincide.

To illustrate what is meant by internal regularity within an algorithm, consider the description of the incoherent combination given in Figure 3-2. The definition of INCOHERENT-COMBINATION is provided implicitly through the alias to the composition of operations shown in Figure 3-6. These operations exhibit a highly regular internal structure. In particular, the sequences feeding into the BANK-OF-SEQUENCES are similar: each adds up the shifted magnitude of the input sequences. By placing a "correspondence constraint" on the sequences feeding into the BANK-OF-SEQUENCES, the manipulation of these expressions and their subexpressions are constrained to occur in synchrony. This constraint results, at least conceptually, in the manipulation of

$$\begin{aligned} Y[n_1, n_2] &= A_{(n_2+1)}[n_1] \\ A_i[n] &= S_{1,i}[n] + S_{2,i}[n] + \ldots \\ S_{1,i}[n] &= M_{1,i}[n+N] \\ S_{2,i}[n] &= M_{2,i}[n+2N] \\ &\vdots \end{aligned}$$

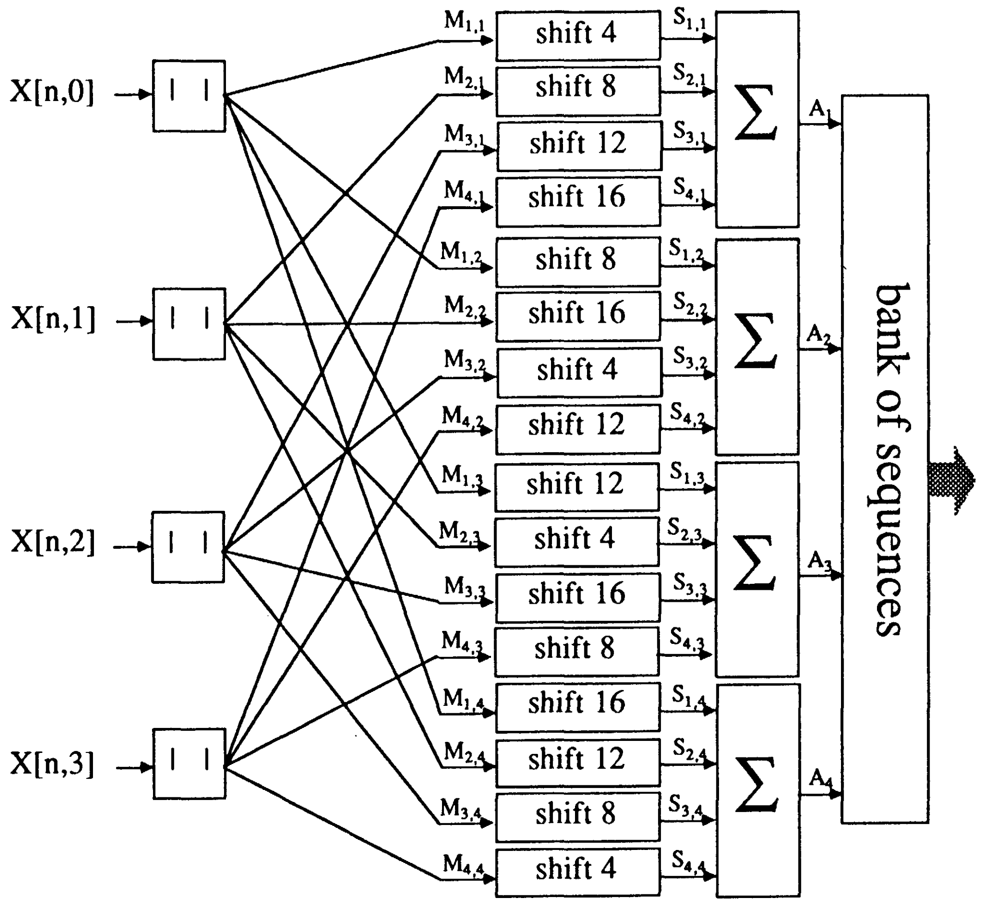

X[n,0]
X[n,1]
X[n,2]
X[n,3]
$M_{1,1}$ shift 4 $S_{1,1}$
$M_{2,1}$ shift 8 $S_{2,1}$
$M_{3,1}$ shift 12 $S_{3,1}$
$M_{4,1}$ shift 16 $S_{4,1}$
$M_{1,2}$ shift 8 $S_{1,2}$
$M_{2,2}$ shift 16 $S_{2,2}$
$M_{3,2}$ shift 4 $S_{3,2}$
$M_{4,2}$ shift 12 $S_{4,2}$
$M_{1,3}$ shift 12 $S_{1,3}$
$M_{2,3}$ shift 4 $S_{2,3}$
$M_{3,3}$ shift 16 $S_{3,3}$
$M_{4,3}$ shift 8 $S_{4,3}$
$M_{1,4}$ shift 16 $S_{1,4}$
$M_{2,4}$ shift 12 $S_{2,4}$
$M_{3,4}$ shift 8 $S_{3,4}$
$M_{4,4}$ shift 4 $S_{4,4}$
Σ
A_1
A_2
A_3
A_4
bank of sequences

Figure 3-6: An example of a signal processing expression with a highly regular internal structure

The internal regularity of signal processing expressions is proposed as an avenue by which the combinatorial explosion of the design space can be avoided. The expression shown here provides an example of what is meant by internal regularity. This expression is the sequence to which the expression (OUTPUT-OF (INCOHERENT-COMBINATION 4 MOD-N+1) X) would alias.

$$\begin{aligned} M_{1,i}[n] &= |X[n, p(N,i,1) - 1]| \\ M_{2,i}[n] &= |X[n, p(N,i,2) - 1]| \\ &\vdots \end{aligned}$$

That is, for $i = 1, ..., N$, the manipulation of $A_i[n]$ occurs in synchrony; the manipulation of $S_{1,i}[n]$ occurs in synchrony, the manipulation of $S_{2,i}[n]$ occurs in synchrony, and so on for the remaining shift-system outputs; the manipulation of $M_{1,i}[n]$ occurs in synchrony, the manipulation of $M_{2,i}[n]$ occurs in synchrony, and so on for the remaining magnitude-system outputs; and the manipulation of $X[n, p(N,i,1) - 1]$ occurs in synchrony, the manipulation of $X[n, p(N,i,2) - 1]$ occurs in synchrony, and so on for the remaining inputs. By enforcing this correspondence, the number of independently manipulated subexpressions is reduced from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$.

In addition, the structure shown in Figure 3-6 shows another point of regularity, originating at the addition operations. In particular, the sequences feeding into each of the SEQUENCE-ADD systems are similar: the first addend into the i'th summation is similar to the second addend into the i'th summation is similar to the N'th addend into the i'th summation. Thus, a second set of correspondence constraints is placed on the inputs into each of the SEQUENCE-ADD systems. These two levels of constraints conceptually result in the manipulation of

$$\begin{aligned} Y[n_1, n_2] &= A_{n_2+1}[n_1] \\ A_i[n] &= \sum_{k=1}^{N} S_{i,k}[n] \\ S_{i,k}[n] &= M_{i,k}[n + kN] \\ M_{i,k}[n] &= |X[n, p(N,i,k) - 1]| \end{aligned}$$

That is, for $i = 1, ..., N$, the manipulation of $A_i[n]$ occurs in synchrony and, for $i = 1, ..., N$ and $k = 1, ..., N$, the manipulation of $S_{i,k}[n]$ occurs in synchrony; the manipulation of $M_{i,k}[n]$ occurs in synchrony; and the manipulation of $X[n, p(N,i,k) - 1]$ occurs in synchrony. These two levels of constraints are imposed on the low-level signal-processing description by the INCOHERENT-COMBINATION system definition. By enforcing these constraints, the number of independently manipulated subexpressions is reduced from $\mathcal{O}(N^2)$ to $\mathcal{O}(N^0) = \mathcal{O}(1)$.

This approach of imposing correspondence constraints and manipulating the corresponding expressions and subexpressions in synchrony is used in ADE to limit the search space for algorithm design.

3.4 Application of ADE to the Design of the FSK-code Detector for Multiple-beam Sonar

The FSK-code detector, described earlier in this chapter, separates naturally into three subproblems: the recovery of the in-phase and quadrature samples of the sonar return; the modulated filter bank, for matched-filter detection of the individual frequency chips; and the incoherent combinations of the filter-bank outputs, to create the detectors for the full FSK code set. The problem of I- and Q-sample recovery will be considered in Chapter 8. When the incoherent combinations of the filter-bank outputs were analyzed using ADE to find equivalent implementations, the only alternatives to the original expression which were found was the master signal of the original expression and a similar expression with the common shift pulled outside the SEQUENCE-ADD and BANK-OF-SEQUENCES operations. This lack of alternatives is explained by examining the computations used in the incoherent detectors. None of the detectors share a common partial sum and no other, more efficient methods are available for taking the magnitude of the input sequences.

This section examines the result of using the Algorithm Design Environment to find alternate implementations of the modulated filter bank.

3.4.1 Matched filtering for the individual frequency chips using general N-point windows

The bank of matched filters in Figure 3-1 uses a general N-point window to shape the individual frequency chips. Figure 3-7 shows the use of ADE to uncover alternate implementations for a general 16-point FIR modulated filter bank.[3]

Line *I-2* of Figure 3-7 requests a list of all the alternate implementations for the modulated filter bank which can be uncovered using constrained manipulations. Line *O-2* of Figure 3-7 shows some of the uncovered equivalent forms. Besides the given form of the modulated filter bank, the Fourier- and z-domain representations of the modulated filter bank were found as was the FFT-based structure shown in Figure 3-8.

Line *I-3* of Figure 3-7 requests the list of the alternate implementations, obtained using constrained manipulations, which are computationally efficient.[4] The most efficient implementation which was found is shown in Figure 3-8.

The implementations which were found in this search span a wide range of structural forms. The modulated filter bank is described in the discrete-time domain, the Fourier domain and the z domain. A completely different implementation for the modulated filter bank is provided in the use of the short-time Fourier transform. This implementation exploits the fact that the impulse responses in the filter bank are related by the modulation factors of $e^{-j\frac{2\pi}{N}kn}$. In addition to describing the short-time Fourier transform explicitly, using the SHORT-TIME-FT system, ADE also provides the expansion of this operation into its component shifts, scales and additions.

Although these implementations span a wide range of structural forms, this result is deceptive. The deceptive quality of this example arises from the fact that one of the equivalence rules which was explicitly given to the environment provides the transformation from the modulated filter bank to the short-time Fourier transform: this rule

[3]Some of the expressions obtained in these and subsequent searches for equivalent forms and efficient implementations are only described verbally or mathematically, due to the unwieldy size of the LISP descriptions.

[4]The way that ADE measures computational efficiency will be described in Chapter 7.

I-1 ADE: (NAMED-SETQ W (A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE
&PROPERTIES :NON-ZERO-SUPPORT [0 16])
X (A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE))

O-1 $\Longrightarrow$ #<X>

I-2 ADE: (CONSTRAINED-EQUIVALENT-FORMS
(OUTPUT-OF (OUTPUT-OF (MODULATED-FILTER-BANK (REVERSE W) 16) X))

O-2 $\Longrightarrow$ (#<(OUTPUT-OF (MODULATED-FILTER-BANK (REVERSE W) 16) X)>
#<(BANK-OF-SEQUENCES (INVERSE-FOURIER-TRANSFORM ...) ...)>
; (BANK-OF-SEQUENCES IFT_0 ... IFT_{15}*)*
; where $IFT_k = \mathcal{F}^{-1}\{\mathcal{F}\{X\}FT\_W(e^{-j\omega+j\frac{\pi}{8}k})\}$
; where $FT\_W = \mathcal{F}\{W\}$
#<(BANK-OF-SEQUENCES (INVERSE-Z-TRANSFORM ...) ...)>
; (BANK-OF-SEQUENCES IZT_0 ... IZT_{15}*)*
; where $IZT_k = \mathcal{Z}^{-1}\{\mathcal{Z}\{X\}ZT\_W(z^{-1}e^{j\frac{\pi}{8}k})\}$
; where $ZT\_W = \mathcal{Z}\{W\}$
#<(OUTPUT-OF (SHORT-TIME-FT W 16) X)>
#<(BANK-OF-SEQUENCES (SEQUENCE-ADD ...) ...)>
; (BANK-OF-SEQUENCES $FT16_0$ ... $FT16_{15}$*)*
; where $FT16_k = FT8_{2*k} + e^{-j\frac{2\pi}{16}k}FT8_{2*k+1}$ *for* $k = 0, ..., 7$
; where $FT16_{k+8} = FT8_{2*k} - e^{-j\frac{2\pi}{16}k}FT8_{2*k+1}$ *for* $k = 0, ..., 7$
; where $FT8_{2*k+l} = FT4_{4*k+l} + e^{-j\frac{2\pi}{8}k}FT4_{4*k+l+2}$ *for* $k = 0, ..., 3$ *and* $l = 0, 1$
; where $FT8_{2*k+l+8} = FT4_{4*k+l} - e^{-j\frac{2\pi}{8}k}FT4_{4*k+l+2}$ *for* $k = 0, ..., 3$ *and* $l = 0, 1$
; where $FT4_{4*k+l} = FT2_{8*k+l} + e^{-j\frac{2\pi}{4}k}FT2_{8*k+l+4}$ *for* $k = 0, 1$ *and* $l = 0, ..., 3$
; where $FT4_{4*k+l+8} = FT2_{8*k+l} - e^{-j\frac{2\pi}{4}k}FT2_{8*k+l+4}$ *for* $k = 0, 1$ *and* $l = 0, ..., 3$
; where $FT2_l = w[l]x[n+l] + w[l+8]x[n+l+8]$ *for* $l = 0, ..., 7$
; where $FT2_{l+8} = w[l]x[n+l] - w[l+8]x[n+l+8]$ *for* $l = 0, ..., 7$
...) *; additional forms*

I-3 ADE: (CONSTRAINED-EFFICIENT-IMPLEMENTATIONS
(OUTPUT-OF (MODULATED-FILTER-BANK W 16) X))

O-3 $\Longrightarrow$ (#<(OUTPUT-OF (SHORT-TIME-FT W 16) X)>
#<(BANK-OF-SEQUENCES (SEQUENCE-ADD ...) ...)>)
; general FFT structure, described mathematically above

Figure 3-7: Manipulation of the general N-point matched filter bank in ADE

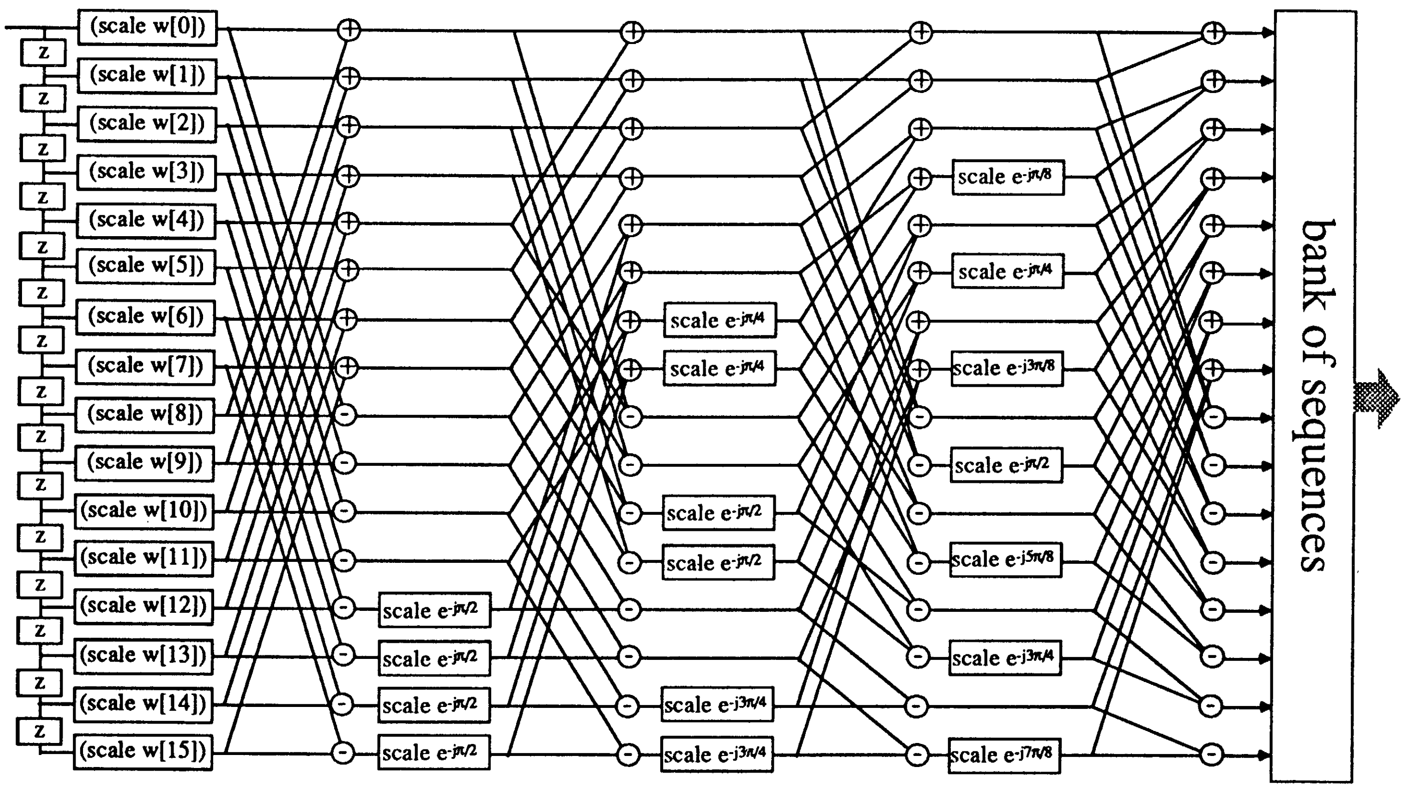


Figure 3-8: One of the equivalent forms found using constrained manipulations on the general N-point FIR matched filter

```
1  (DEFINE-SYSTEM-CLASS-ALIAS
2      (MODULATED-FILTER-BANK
3        IMPULSE-RESPONSE@DISCRETE-TIME-SEQUENCE N@INTEGER
4        &OPTIONAL (DOWNSAMPLING-FACTOR@INTEGER 1))
5          ; accept a sequence and an integer as system parameters
6      (INPUT@DISCRETE-TIME-SEQUENCE)      ; accept a sequence as an input
7          (SHIFT-INVARIANT-SYSTEM LINEAR-SYSTEM 2D-SYSTEM)
8          ; a subclass of these classes
9    ("a modulated filter bank"
10     SELF)      ; the systems "alias" only themselves
11   NIL ()       ; generate a new output signal class, without any additional superior signal classes
12   ("the output from a modulated filter bank"
13       ; the output signals "alias" this composition of operations
14     (MAP-OVER 'BANK-OF-SEQUENCES I 0 N
15       (OUTPUT-OF (DOWNSAMPLE DOWNSAMPLING-FACTOR)
16         (SEQUENCE-CONVOLVE
17           (SEQUENCE-MULTIPLY
18             (COMPLEX-EXPONENTIAL-SEQUENCE (/ (* 2PI I) N))
19             IMPULSE-RESPONSE)
20           INPUT)))
21     (GOAL EQUIVALENT-FORM       ; the short-time Fourier transform implementation
22       :NAME AS-STFT
23       :OBJECT ?SELF       ; any modulated filter bank.
24       :ANSWER (OUTPUT-OF (SHORT-TIME-FT (REVERSE IMPULSE-RESPONSE) N
25                                          DOWNSAMPLING-FACTOR)
26                 INPUT))))
```

Figure 3-9: The definition for the system class, modulated-filter-bank

was included in the definition of the modulated filter bank, as shown on lines 21–26 of Figure 3-9.

This example actually points out one of the important advantages of an algorithm design environment: the design environment allows known transformations to be encoded and, thereafter, the environment itself will to remember and apply these transformations. It is exactly this process of encoding and automatic application that has been seen in this example.

In addition, this example illustrates the usefulness of regularity constraints in algorithm manipulation. If correspondence constraints were not used to restrict the combinations of subexpressions, the five alternate implementations of a finite-length convolution would result in the consideration of more than 10^{11} equivalent forms, none of which are significantly different in structure from the modulated filter bank itself. With the correspondence constraints, these five alternate implementations result in the consideration of only five equivalent forms, since the manipulation of the convolutions is constrained to occur in synchrony. Similarly, without parallel manipulations, combinatorial growth will also occur with the FFT-based structures.

3.4.2 Matched filtering for the individual frequency chips using N-point rectangular windows

The selection of the frequency-chip window affects both the range resolution and the signal-to-signal rejection of the sonar system. The autocorrelation of the frequency-chip window dictates the shape of the responses of the individual filters to the matched frequency chips. The signal-to-signal rejection dictates the drop-off in the responses of the matched filters as they travel across the FSK-code echo and encounter the mismatched frequency chips. A simple choice for the frequency-chip window is the N-point rectangular window. As seen in Figure 3-10, the rectangular window has a sharply peaked autocorrelation and good signal-to-signal rejection.

Figure 3-11 shows use of ADE to uncover alternate implementations for the modulated filter bank using a 16-point rectangular window. Line *I-2* of Figure 3-11 requests a list of all the alternate implementations for the modulated rectangular-window filter bank which can be obtained using constrained manipulations. Line *O-2* of Figure 3-11 shows some the uncovered equivalent forms. As with the general N-point modulated filter bank, the Fourier- and z-domain representations of the modulated filter bank were found as was the general FFT-based structure shown in Figure 3-8. In addition to these structures, a variety of "pruned" FFT-based structures were uncovered: one of these structures is shown in Figure 3-12. These implementations have the same underlying structure as the general FFT implementation shown in Figure 3-8. The difference lies in the *number* of

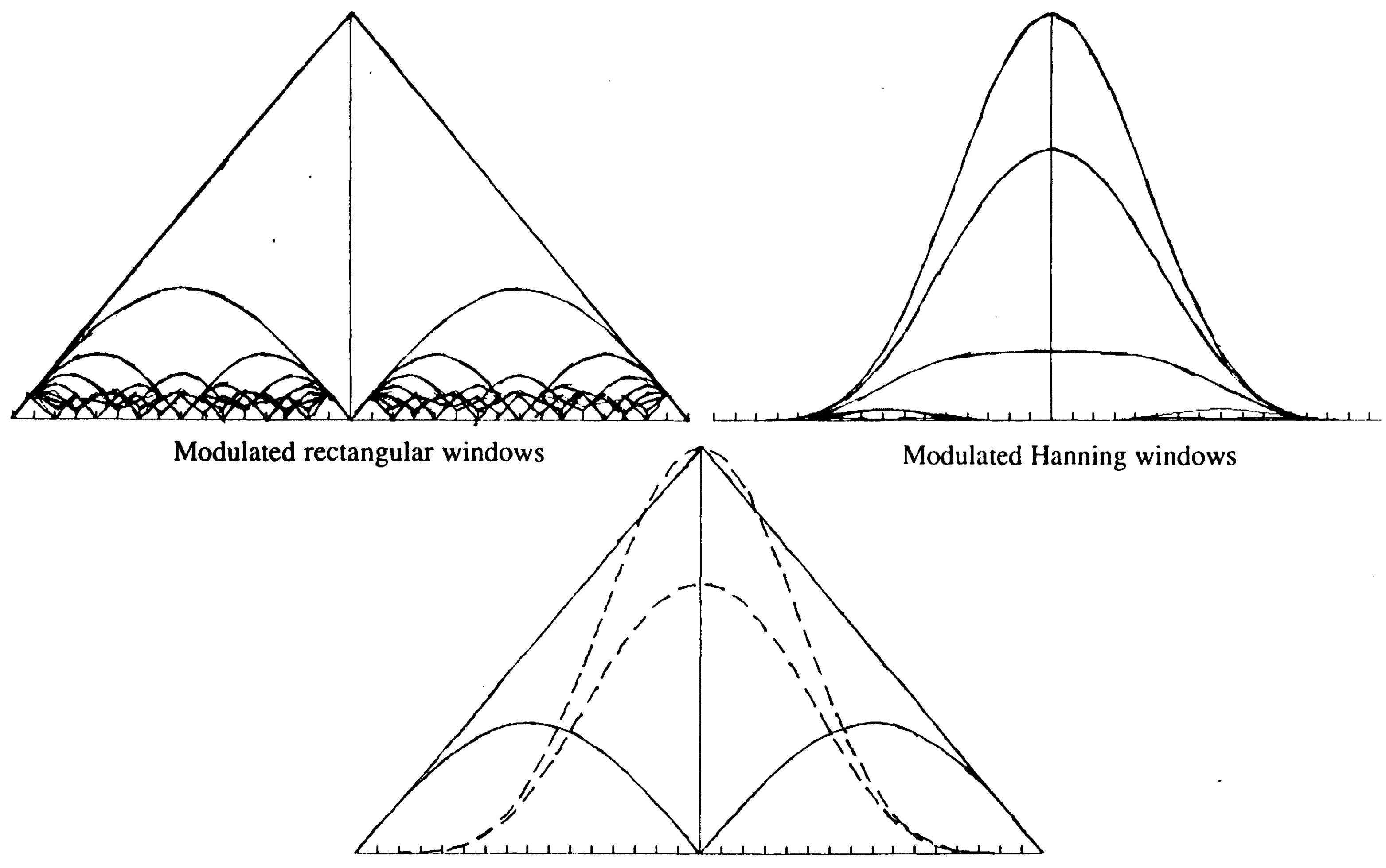


Figure 3-10: The amplitudes of the autocorrelations and cross-correlations of the 16-point modulated rectangular and Hanning windows

The performance of the sonar system is strongly affected by the autocorrelation and the cross-correlation of the frequency-chip window. In particular, the resolution both in range and in azimuth and elevation is dictated by the sharpness of the autocorrelation and by signal-to-signal rejection of the modulated windows. This figure compares the auto- and the cross-correlations of the rectangular windows and of the Hanning windows. Although the autocorrelation peak of the rectangular window is wider than that of the Hanning window, the signal-to-signal rejection is significantly better.

```
I-1 ADE: (NAMED-SETQ RECTANGULAR (REVERSE (CAUSAL-RECTANGULAR-SEQUENCE 16))
                      X (A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE))
O-1 ⟹ #<X>
I-2 ADE: (CONSTRAINED-EQUIVALENT-FORMS
           (OUTPUT-OF (MODULATED-FILTER-BANK RECTANGULAR 16) X))
O-2 ⟹ (#<(OUTPUT-OF (MODULATED-FILTER-BANK RECTANGULAR 16) X)>
        #<(BANK-OF-SEQUENCES (INVERSE-FOURIER-TRANSFORM ... ) ... )>
                      ; Fourier-domain implementation of filter bank
        #<(BANK-OF-SEQUENCES (INVERSE-Z-TRANSFORM ... ) ...)>
                      ; z-domain implementation of filter bank
        #<(OUTPUT-OF (SHORT-TIME-FT RECTANGULAR 16) X)>
        #<(BANK-OF-SEQUENCES (SEQUENCE-ADD ... ) ... )>
                      ; general FFT structure
        #<(BANK-OF-SEQUENCES (SEQUENCE-ADD ... ) ...)>
                      ; (BANK-OF-SEQUENCES $FT16_0$ ... $FT16_{15}$)
                      ;     where $FT16_k[n] = FT8_k[n] + e^{-j\frac{2\pi}{16}k} FT8_k[n+1]$ for $k = 0, ..., 7$
                      ;     where $FT16_{k+8}[n] = FT8_k[n] - e^{-j\frac{2\pi}{16}k} FT8_k[n+1]$ for $k = 0, ..., 7$
                      ;     where $FT8_k[n] = FT4_k[n] + e^{-j\frac{2\pi}{8}k} FT4_k[n+2]$ for $k = 0, ..., 3$
                      ;     where $FT8_{k+4}[n] = FT4_k[n] - e^{-j\frac{2\pi}{8}k} FT4_k[n+2]$ for $k = 0, ..., 3$
                      ;     where $FT4_k[n] = FT2[n] + e^{-j\frac{2\pi}{4}k} FT2[n+4]$ for $k = 0, 1$
                      ;     where $FT4_{k+2}[n] = FT2[n] + -e^{-j\frac{2\pi}{4}k} FT2[n+4]$ for $k = 0, 1$
                      ;     where $FT2[n] = x[n] + x[n+8]$
                      ;     where $FT2[n] = x[n] - x[n+8]$
        ... )     ; additional forms
I-3 ADE: (CONSTRAINED-EFFICIENT-IMPLEMENTATIONS
           (OUTPUT-OF (MODULATED-FILTER-BANK RECTANGULAR 16) X))
O-3 ⟹ (#<(BANK-OF-SEQUENCES (SEQUENCE-ADD ... ) ... )>
                      ; general FFT structure
        #<(BANK-OF-SEQUENCES (SEQUENCE-ADD ... ) ... )>
                      ; pruned FFT structure, described mathematically above
        ... )     ; additional forms
```

Figure 3-11: Manipulation of the modulated rectangular-window filter bank in ADE

butterflies that are computed at each stage. For example, the pruned FFT structure shown in Figure 3-12 has only one butterfly in the first stage, two in the second, four in the third and eight in the fourth while the general FFT structure shown in Figure 3-8 has eight butterflies in each stage.

Line *I-3* of Figure 3-11 requests the list of the alternate implementations, obtained using constrained manipulations, which are computationally efficient. Of the thirteen equivalent implementations which were found using constrained manipulations, only the general FFT structure (Figure 3-8 with $w[n]$ constrained to be a 16-point rectangular window) and the pruned FFT structures (like Figure 3-12) were found to be efficient. As can be seen from the comparison of costs shown in Figure 3-12, a tradeoff exists between minimum number of memory locations, achieved by the general FFT structure, and the minimum number of operation counts, achieved by the pruned FFT structure.[5]

It is interesting to note that, with the pruned FFT, the *order* of the computational complexity is actually reduced as well as the number of computations themselves. The order is reduced from $\mathcal{O}(N^2)$ for the direct-form implementation or from $\mathcal{O}(N \log N)$ for the general FFT implementation to $\mathcal{O}(N)$ for the pruned FFT implementation. The amount of computation which is required for the pruned FFT is actually identical to that of the recursive computation of the sliding Fourier transform:[6]

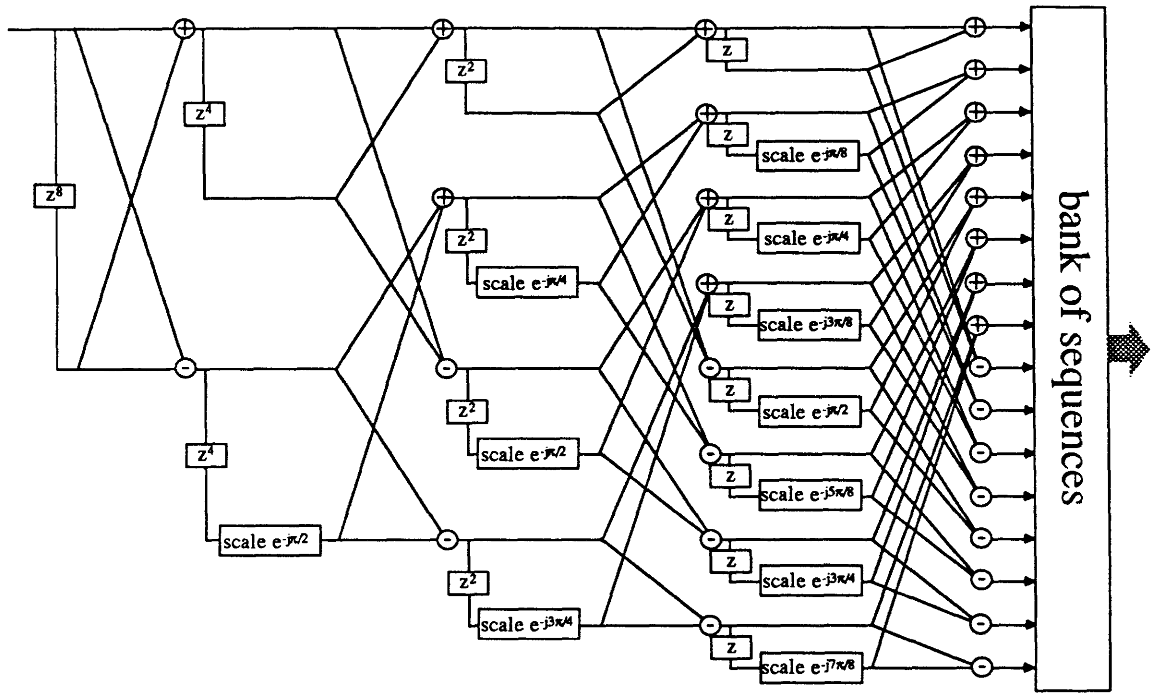
$$X[n,k] = e^{j\frac{2\pi}{N}k}(X[n-1,k] + x[n+N-1] - x[n-1])$$

For reasons which will be discussed in Chapter 8, ADE is unable to represent this explicitly recursive formulation of the computation.

The pruned FFT structure shown in Figure 3-12 has not been found in the currently published literature: the only reference with a similar structure is Regalia (1989) which has been submitted for publication in *SIAM Review.* Although other pruned FFT struc-

[5]The memory counts do not include the registers necessary for storing the intermediate sequence values. If these additional memory locations were included in the cost structures, the amount of memory for the general FFT structure using the method given by Singleton (1969) and the pruned FFT structure would be identical.

[6]The pruned FFT structure has the advantage of being numerically stable while the recursive formulation is unstable due to its reliance on pole/zero cancellation on the unit circle.



	Computational costs for N = 16			Approximate computational costs for general N		
Structure	complex multiplies	complex adds	memory locations	complex multiplies	complex adds	memory locations
modulated filter bank	256	240	512	N (N - 1)	N	$2\,N^2$
general FFT structure	17	64	64	$N \log_2 N$	$\frac{N}{2} \log_2 \frac{N}{2}$	3 N - 1
pruned FFT structure	11	30	86	2 N	N	$N \log_2 N + N$

Figure 3-12: One of the equivalent forms found using constrained manipulation on the matched filters for the modulated rectangular windows

Some of the equivalent forms obtained for the modulated filter bank of the rectangular windows had a basic FFT structure with some of the butterflies removed from the early stages. One of these "pruned" FFT structures is shown here. A comparison of the costs of some of the alternate implementations is shown as well.

tures have been published (Markel, 1971; Skinner, 1976), these structures depend on the characteristics of the inputs as opposed to the characteristics of the desired outputs.

The actual transformation rule which is crucial for obtaining this implementation is surprisingly simple. As shown on lines 14–33 of Figure 3-13, the crucial transformation rule simply pulls common shifts through a generalized shift-invariant system.[7] By pulling all the common shift operations through the butterfly and twiddle stages, the computational structure collapses from the general FFT structure, shown in Figure 3-8, to the pruned FFT structure, shown in Figure 3-12.

This example also strongly supports the use of constrained manipulations on regular algorithms. Using simple combinatoric analysis, the number of equivalent forms which would be found by unconstrained manipulations can be estimated: this number is more than 10^{19} as opposed to the thirteen found with constrained manipulations. These 10^{19} unconstrained equivalent forms simply use mismatched combinations of the subexpressions used in the thirteen constrained equivalent forms. The severity of this combinatorial growth is due to the branching of the FFT structure. Using unconstrained manipulations, 720 subexpressions go into making up the general 16-point FFT structure shown in Figure 3-8. Without regularity constraints to limit the combination of these expressions and subexpressions, if M distinct, independent transformations are available for each of these subexpressions, then there will be $\mathcal{O}(M^{720})$ distinct equivalent forms to be considered.

3.4.3 Matched filtering for the individual frequency chips using N-point Hanning windows

Since the FSK-code signals are created from sequences of contiguous frequency chips, the input signal to the sonar detector will include energy from these contiguous frequency

[7]The term "generalized" is used here to distinguish these systems from the classic shift-invariant systems. In the classic shift-invariant system, $H_1\{\ \}$, if $y_1(t) = H_1\{x(t)\}$ then $y_1(t-T) = H_1\{x(t-T)\}$. In a generalized shift-invariant system, $H_2\{\ \}$, if $y_2(t) = H_2\{x_1(t), ..., x_N(t)\}$ then $y_2(t-T) = H_2\{x_1(t-T), ..., x_N(t-T)\}$.

```
(DEFINE-ABSTRACT-SYSTEM-CLASS (GENERALIZED-SHIFT-INVARIANT-SYSTEM *) *
          ; accept any parameters or inputs
    ()      ; no superclasses
  ("H{ } s.t. if y[n] = H{x1[n] ... xL[n]} then y[n-N] = H{x1[n-N] ... xL[n-N]}")
  NIL ()      ; generate a new output signal class, without any additional superior signal classes
  ("the output from a generalized shift-invariant system"
    (GOAL SIMPLIFICATION      ; if all the inputs are shifted identically, pull shift system outside
      :NAME SHIFTED-INPUT
      :OBJECT (OUTPUT-OF ?SYSTEM@(NOT SHIFT-SYSTEM) &REST
                 ?INPUTS${(OUTPUT-OF ?SHIFT ?[SHIFT-INPUTS])})
                              ; the inputs are all outputs from a single shift system.
                              ; ?SHIFT-INPUTS will be bound to the list of inputs to the shift system.
      :ANSWER (OUTPUT-OF SHIFT (APPLY 'OUTPUT-OF SYSTEM SHIFT-INPUTS)))
    (GOAL EQUIVALENT-FORM      ; if all the inputs are shifted, pull one of the shifts outside
      :NAME UNEQUALLY-SHIFTED-INPUT
      :OBJECT (OUTPUT-OF ?SYSTEM@(NOT SHIFT-SYSTEM) &REST
                 ?INPUTS${(OUTPUT-OF (SPECIFIC-MEMBER SHIFT-SYSTEM
                                        &REST ?[SHIFT-FACTORS])
                              ?[SHIFT-INPUTS])})
                                ; the inputs are all outputs from shift systems.
                                ; ?SHIFT-FACTORS will be bound to the list of the amounts of the shifts.
                                ; ?SHIFT-INPUTS will be bound to the list of inputs to the shift system.
      :ANSWER
      (LET ((COMMON-SHIFT (FIRST SHIFT-FACTORS)))      ; pull first shift outside
        (OUTPUT-OF (APPLY 'SHIFT COMMON-SHIFT)
          (APPLY 'OUTPUT-OF SYSTEM
                 (MAPCAR
                   '(LAMBDA (SHIFT-FACTOR SHIFT-INPUT)
                      (OUTPUT-OF (APPLY 'SHIFT
                                          (MAPCAR '$-      ; compensate for outside shift
                                            SHIFT-FACTOR COMMON-SHIFT))
                        SHIFT-INPUT))
                   SHIFT-FACTORS SHIFT-INPUTS)))))
    ... ))
```

Figure 3-13: The definition for the system class, generalized-shift-invariant-system

chips. As shown in Figure 3-14, the 16-point Hanning window is actually preferable to the 16-point rectangular window in terms of the worst-case responses of the matched filters to a sequence of contiguous frequency chips: the worst-case side-lobe heights are essentially identical and the worst-case main-lobe width of the Hanning-window filters is approximately a third of that for the rectangular-window filters. ADE was again used to find the constrained equivalent forms and the constrained efficient implementations of the modulated filter bank using the 16-point Hanning window.[8] In response to the request for constrained equivalent forms, the same set of structures which were found for the general modulated filter bank were again obtained. In addition, structures like the one shown in Figure 3-15 were uncovered. These algorithms are based on the pruned FFT implementations of the rectangular window with the application of the Hanning window occurring in the discrete Fourier domain. These pruned FFT implementations for the Hanning window were among the efficient implementations of the modulated Hanning-window filter bank.

As with the modulated rectangular-window filter bank, this example indicates the potential of automated algorithm manipulation. This example also supports the use of constrained search: again using simple combinatoric analysis, the number of equivalent forms which would be found by unconstrained manipulations would be more than 10^{58} as opposed to the twenty found with constrained manipulations.[9] Furthermore, this example is interesting for the path by which the structure shown in Figure 3-15 is found.

The new transformation step which must be completed to derive this structure goes from the Hanning-window, short-time Fourier transform to the rectangular-window, short-time Fourier transform followed by circular convolution. The other sections of

[8]The actual input and output are not shown since the format is the same as was shown in Figures 3-7 and 3-11: this repetition was not considered useful.

[9]This number of forms found using constrained manipulations is actually artificially low, since the author interrupted the search and added a correspondence constraint to force coincidence between the implementations of the rectangularly-windowed short-time Fourier transforms. If the environment had been allowed to continue with the constrained search undisturbed, approximately two thousand equivalent forms would have been found. This example presents a strong argument for having the environment itself look for correspondences within signal-processing expressions, so that this artificial intervention would not be necessary. This is one of the suggestions made in Chapter 8 for future research.

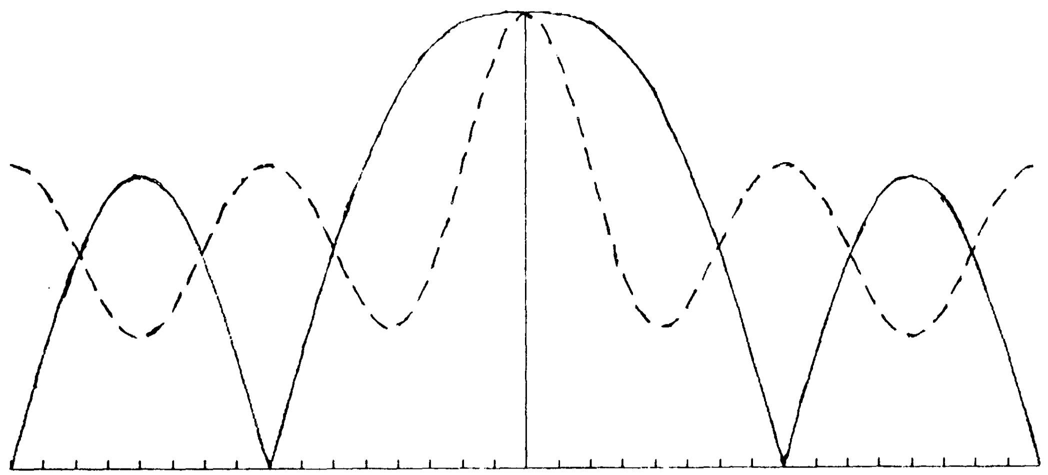


Figure 3-14: Part of the amplitude responses of the 16-point rectangular-window filters and of the 16-point Hanning-window filters to a sequence of frequency chips

The amplitude response of the matched filters to the sequence of 16 contiguous frequency chips dictates the overall performance of the sonar imaging system. "Worst-case" analyses of the responses of the individual matched filters to a sequence of contiguous frequency chips for both the rectangular and the Hanning windows are shown here. These plots were created by incoherently combining the filter responses to contiguous frequency chips where the modulation of the center chip is correctly matched and the modulations of all the other chips are mismatched by $\Delta\omega = \pm\frac{2\pi}{16}$

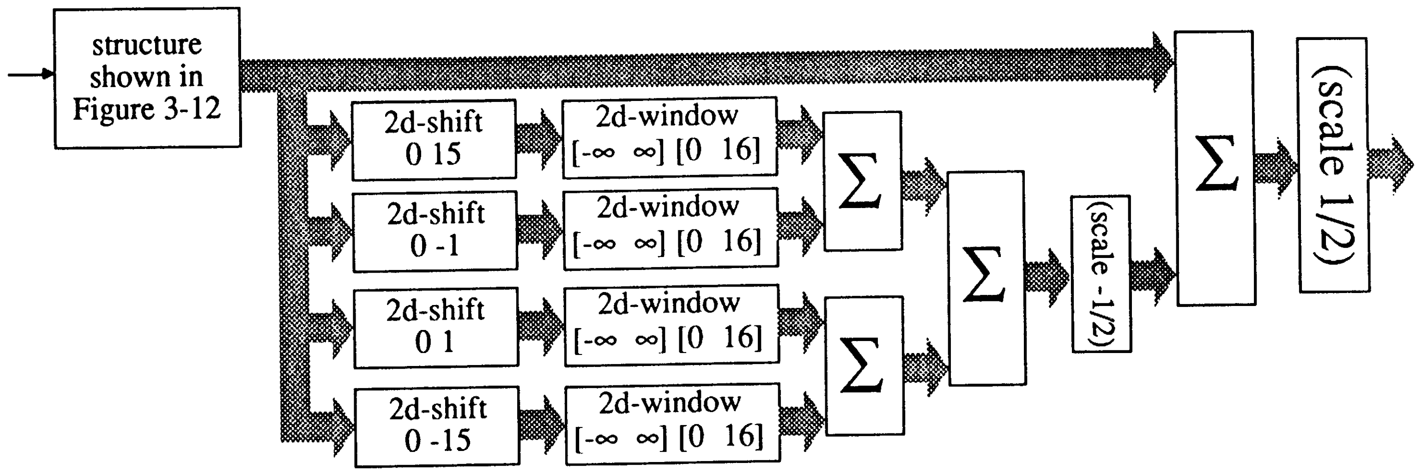


Figure 3-15: One of the equivalent forms found using constrained manipulation on the matched filters for the modulated N-point Hanning windows

Some of the equivalent forms obtained for the modulated filter bank of the N-point Hanning windows used the pruned FFT implementations of the rectangular window and then completed the Hanning-window shaping in the discrete Fourier domain. One of these structures is shown here.

the path from the modulated filter bank to the structure shown in Figure 3-15 have already been traveled. In particular, the transformation from the modulated filter bank to the Hanning-window, short-time Fourier transform is achieved via the rule shown on lines 21–26 of Figure 3-9 and the transformation from the rectangular-window, short-time Fourier transform to the pruned FFT structure was discussed in the previous subsection.

The transformation from the Hanning-window, short-time Fourier transform to the rectangular-window, short-time Fourier transform actually occurs by way of the FFT system. One of the transformation rules included in the definition of the short-time Fourier transform maps the problem of finding the equivalent forms of the short-time Fourier transform into an associated problem using the FFT (lines 26–38 of Figure 3-16). This mapping exploits the fact that the L-point short-time Fourier transform, $Y[n,k] = \sum_m w[m]x[n+m]e^{-j\frac{2\pi}{L}km}$, is the same as the two-dimensional sequence $x[n_0,k] = X_{n_0}[k]$ where X_{n_0} is the L-point discrete Fourier transform of $w[n]x[n+n_0]$. Thus, the search for the equivalent forms of the 16-point, Hanning-window, short-time Fourier transform also attempts to find the equivalent forms of the 16-point FFT of the Hanning window multiplied by some abstract, discrete-time sequence. A new abstract discrete-time sequence must be used in the product since it is not the input sequence, #<x>, which must be represented but rather a whole range of shifted versions of #<x>. It is in the search for the equivalent forms of this 16-point FFT that the transformation from time-domain multiplication to discrete Fourier-domain circular convolution occurs. This transformation was uncovered through the interaction of a number of transformation rules, as shown in Addendum 3.A.

3.4.4 Matched filtering for the individual frequency chips using $2N$-point Hanning windows

Another alternative for a frequency-chip window is the $2N$-point Hanning window. As seen in Figure 3-17-b, for a 16-channel imaging system, an FSK-code signal using 32-point

```
(DEFINE-SYSTEM-CLASS
    (SHORT-TIME-FT WINDOW@DISCRETE-TIME-SEQUENCE FFT-SIZE@INTEGER
      &OPTIONAL (DOWNSAMPLING-FACTOR@INTEGER 1))
        ; accept a sequence and two integers as parameters. the second integer is optional.
    (INPUT@DISCRETE-TIME-SEQUENCE)      ; accept a sequence as an input.
        (LINEAR-SYSTEM 2D-SYSTEM)      ; a subclass of these classes
  ("the short-time Fourier transform system"
    (RECREATE-2D-SEQUENCE (1D-SEQUENCE TOKEN-INPUT REPLACED-INPUT)
      ... ))      ; the back-translation from the 1D FFT to the 2D STFT
  NIL ()      ; generate a new output signal class, without any additional superior signal classes
  ("a short-time Fourier transform $X[m,k] = \mathrm{FFT}_{\mathrm{FFT\text{-}SIZE}}\{x[n+m]w[n]\}$"
    (GOAL SIMPLIFICATION      ; attempt to pull shifts outside
      :NAME SHIFTED-INPUT
      :OBJECT      ; the input is shifted
      (OUTPUT-OF ?SYSTEM$(SPECIFIC-MEMBER SHORT-TIME-FT ?WINDOW ?FT-SIZE
                                             ?DOWNSAMPLING-FACTOR)
        (OUTPUT-OF (SPECIFIC-MEMBER SHIFT-SYSTEM ?SHIFT) ?SHIFT-INPUT))
      :WHEN ($>= ($ABS SHIFT) DOWNSAMPLING-FACTOR)
          ; the shift on the input is greater than the output downsampling factor
      :ANSWER (MULTIPLE-VALUE-BIND (PRE-SHIFT POST-SHIFT)
                  (SEPARATE-SHIFT-THRU-RATE-CONVERSION
                    SHIFT DOWNSAMPLING-FACTOR)
                (OUTPUT-OF (2D-SEQUENCE-SHIFT POST-SHIFT)
                  (OUTPUT-OF SHORT-TIME-FT
                  (OUTPUT-OF (SEQUENCE-SHIFT PRE-SHIFT) INPUT)))))
    (GOAL EQUIVALENT-FORMS      ; use equivalent 1D FFT problem
      :NAME USING-1D-FFT
      :OBJECT ?SELF      ; any STFT.
      :ANSWER      ; find equivalent forms using the corresponding 1D FFT problem
      (LET ((TOKEN-INPUT (A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE)))
        (LET ((1D-EQUIVALENT-FORMS      ; find the eq forms of the equivalent 1D FFT problem
                (EQUIVALENT-FORMS (OUTPUT-OF (FFT FFT-SIZE)
                                    (SEQUENCE-MULTIPLY WINDOW TOKEN-INPUT)))))
          (LOOP FOR 1D-FORM IN 1D-EQUIVALENT-FORMS
                FOR 2D-FORM = (RECREATE-2D-SEQUENCE
                                (GENERATING-SYSTEM SELF)
                                1D-FORM TOKEN-INPUT INPUT)
                WHEN 2D-FORM COLLECT 2D-FORM))))
    ... ))
```

Figure 3-16: The definition for the system class SHORT-TIME-FT

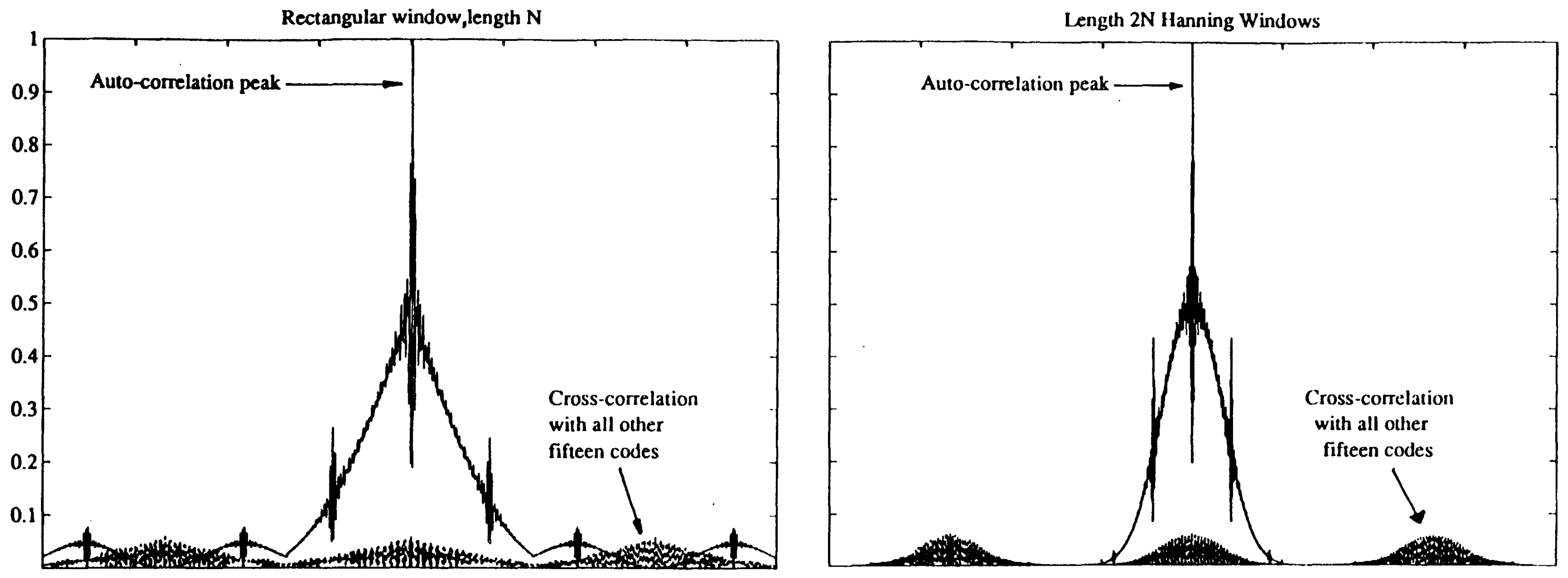


Figure 3-17: The amplitude responses of the FSK-code detectors for signals using 16-point rectangular-window frequency chips and for signals using 32-point Hanning-window frequency chips

Another alternative for a frequency-chip window is the 2N-point Hanning window. The amplitude responses of the detectors are shown here, for the N-point rectangular window and for the 2N-point Hanning window. Even though the reflected energy from the FSK-code using the 2N-point Hanning window extends for twice as long temporally, the range resolution using this longer window is actually better than the resolution using the N-point rectangular window. In addition, the side lobes of the responses are lower using the 2N-point Hanning window.

Hanning windows is preferable to an FSK-code signal using 16-point rectangular windows both in range resolution and in signal-to-signal rejection. The 32-point Hanning windows will also be preferable to the 16-point Hanning windows in signal-to-signal rejection.

ADE was again used to find the constrained equivalent forms and the constrained efficient implementations of the 16-channel, modulated filter bank using the 32-point Hanning window.[10] The same set of structures which were found for the general modulated filter bank were again obtained. In addition to the structures for general modulated filter bank implementations, structures like the one shown in Figure 3-18 were uncovered. As with the N-point Hanning-window structures, these algorithms are based on the pruned FFT implementations of the rectangular window with the application of the Hanning window essentially occurring in the discrete Fourier domain. The striking difference between the structure shown in Figure 3-18 and the N-point Hanning-window structures, is that the structure in Figure 3-18 essentially relies on a $2N$-point FFT: the environment itself obtains the $2N$-point FFT structure from the N-point short-time Fourier transform implementation of the modulated filter bank.

The rule which made this derivation possible is comparatively simple and general: it is shown on lines 14–33 of Figure 3-19 within the definition of the generalized homogeneous system class.[11] The rule extracts common scaling factors and moves them to outside the generalized homogeneous system. The actual sequence of transformations which were traveled through to obtain the structure in Figure 3-18 is shown in Addendum 3.B. As shown in this addendum, the original problem is eventually recast into a problem of finding the equivalent forms of the sum of shifted versions of the short-time Fourier transform using the 16-point rectangular window, $r_{16}[n]$, and of shifted, scaled versions of the short-time Fourier transform using $e^{-j\frac{\pi}{16}n}r_{16}[n]$.[12] The problem of finding the

[10]The actual input and output are not shown since the format is the same as was shown in Figures 3-7 and 3-11: this repetition was not considered useful.

[11]As with the generalized shift-invariant system class, the adjective "generalized" is meant to indicate the extension from a single-input system to a multi-input system. So, with a generalized homogeneous system $H\{\}$, if $y[n] = H\{x_1[n], ..., x_N[n]\}$ then $ay[n] = H\{ax_1[n], ..., ax_N[n]\}$.

[12]The reduction of the original problem to this intermediate problem also relies on the behavior of the FFT-output equivalent-form rule MODULATED-INPUT. In particular, in ADE, this rule is written to reduce

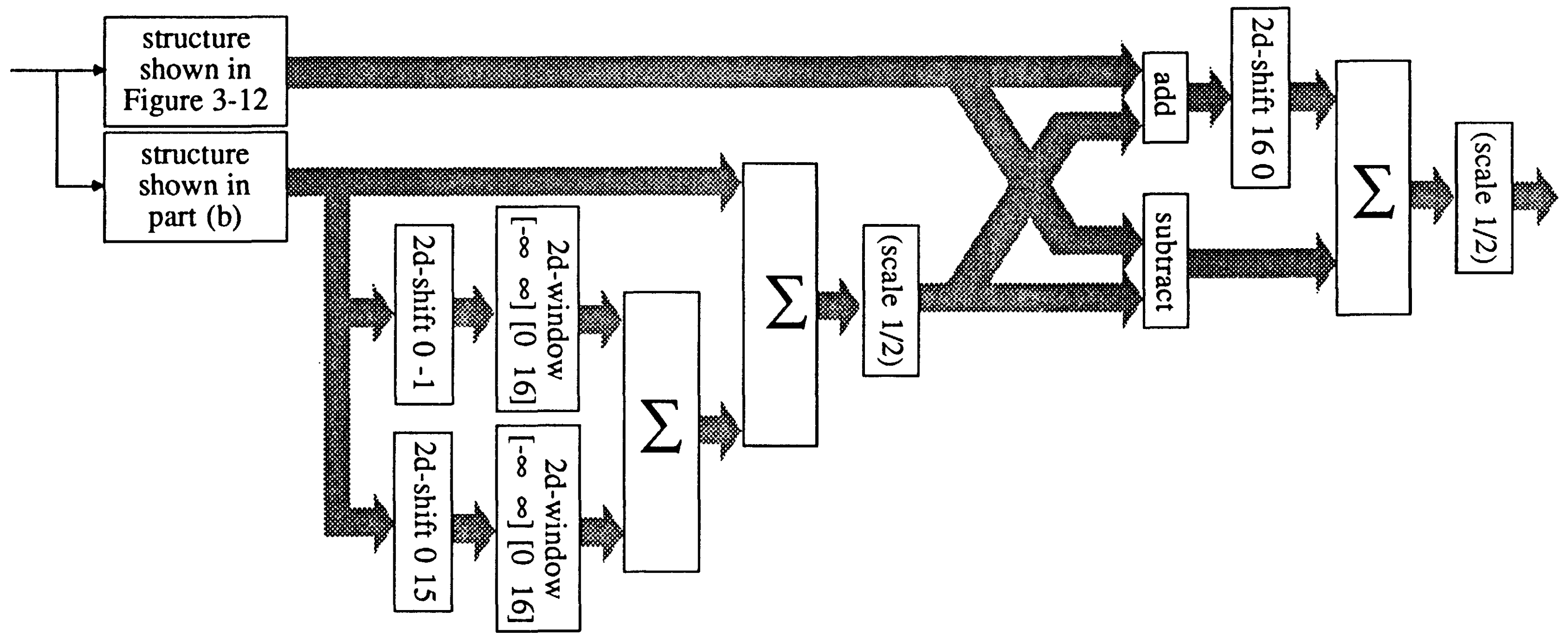
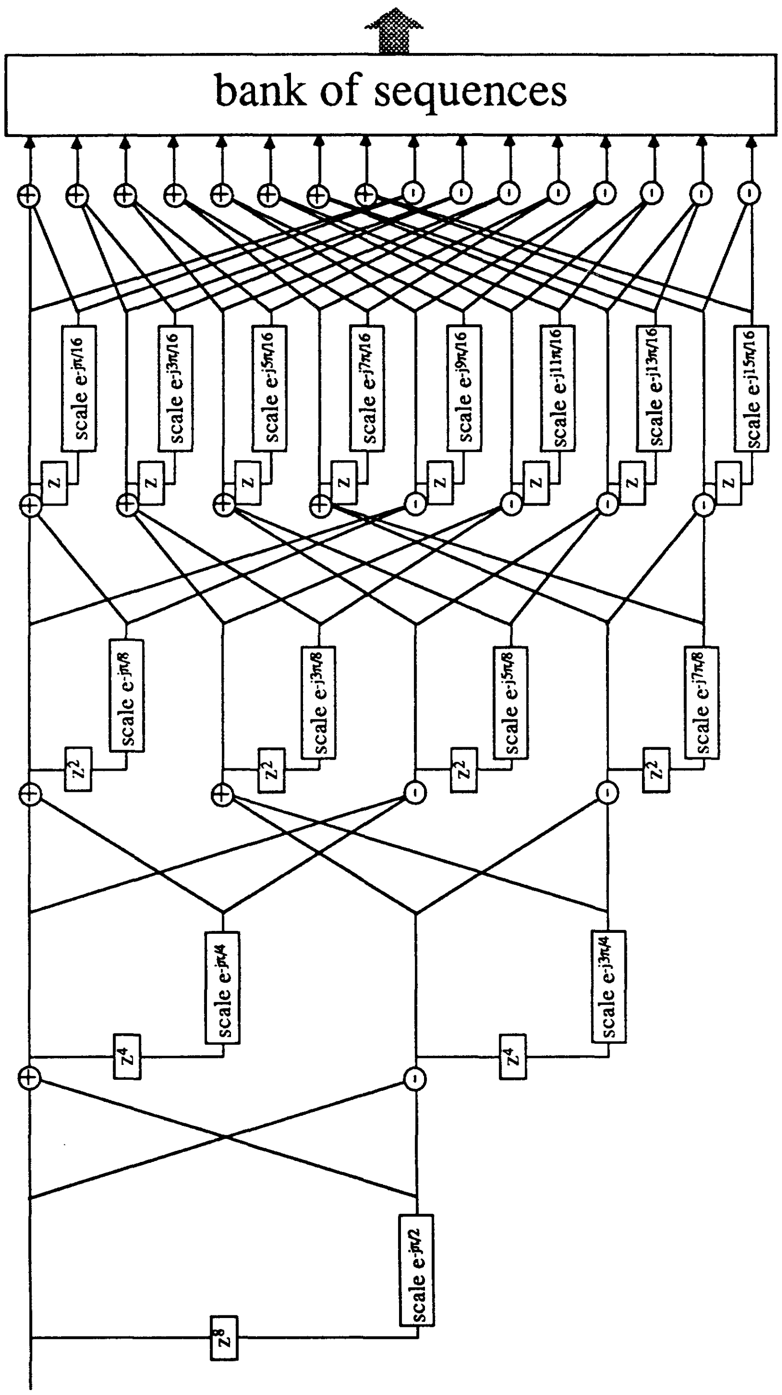


Figure 3-18: One of the equivalent forms found using constrained manipulation on the matched filters for modulated 2N-point Hanning windows

Some of the equivalent forms obtained for the modulated filter bank of the 2N-point Hanning windows in effect used a pruned 2N-point FFT implementation of the 2N-point rectangular window and then completed the Hanning-window shaping in the discrete Fourier domain. One of these structures is shown here. The interesting part of this implementation is that the design environment itself travelled from the N-point short-time Fourier transform structure to the 2N-point Fourier transform structure.

part (b)


bank of sequences
scale $e^{-j\pi/16}$
scale $e^{-j3\pi/16}$
scale $e^{-j5\pi/16}$
scale $e^{-j7\pi/16}$
scale $e^{-j9\pi/16}$
scale $e^{-j11\pi/16}$
scale $e^{-j13\pi/16}$
scale $e^{-j15\pi/16}$
z
scale $e^{-j\pi/8}$
scale $e^{-j3\pi/8}$
scale $e^{-j5\pi/8}$
scale $e^{-j7\pi/8}$
z^2
scale $e^{-j\pi/4}$
scale $e^{-j3\pi/4}$
z^4
scale $e^{-j\pi/2}$
z^8

equivalent forms of the short-time Fourier transform using the rectangular window has already been solved. The problem of finding the equivalent forms of the short-time Fourier transform using the modulated rectangular window uses the rule shown in Figure 3-19 repetitively, to move the common scaling factors of $e^{-j\frac{2\pi}{32}k}$ outward through the general FFT structure shown in Figure 3-8. Once this modulation is moved through the general structure, the same pruning as was seen for the rectangularly windowed, short-time Fourier transform can be completed.

The rule shown in Figure 3-19 was included in ADE in hopes of making this very derivation the $2N$-point FFT structure from the N-point FFT structure. The targeted inclusion of this particular transformation rule had an additional benefit. Specifically, when this rule for pulling common scaling factors outside a generalized homogeneous system was encoded into the environment, the corresponding rule for pulling common shifting factors outside a generalized shift-invariant system was also included. This second rule was envisioned and encoded simply to maintain the parallel in behavior between the two "generalized" system classes. Thus, the derivation of the $2N$-point FFT structure relies on the use of a rule targeted specifically for this discovery. The formulation of this targeted rule resulted in the formulation and inclusion of another, untargeted rule. It was this untargeted rule which uncovered the pruned FFT structure.

3.5 Summary

To conclude this introduction of the Algorithm Design Environment, the general characteristics of the environment have been outlined, using the sonar FSK-code detector as an example. The signal and system representations which are used in ADE closely mimic their mathematical characterization. In particular, the signal- and system-class definitions are hierarchical, allowing common characteristics, such as shift invariance, to

all modulations of the input to an N-point FFT to modulations of the form $e^{-j\frac{2\pi}{N}\Delta\omega n}$ with $0 \leq \Delta\omega < 1$. It is this rule that maps the modulation by $e^{j\frac{2\pi}{32}n}$ into the circularly shifted FFT with a modulation term of $e^{-j\frac{2\pi}{32}n}$.

```
(DEFINE-ABSTRACT-SYSTEM-CLASS (GENERALIZED-HOMOGENEOUS-SYSTEM *) *
            ; accept any parameters or inputs
   ()       ; no superclasses
  ("H{ } s.t. if y[n] = H{x1[n] ... xL[n]} then a*y[n] = H{a*x1[n] ... a*xL[n]}")
  NIL ()       ; generate a new output signal class, without any additional superior signal classes
  ("the output from a generalized homogeneous system"
    (GOAL SIMPLIFICATION        ; if all the inputs are scaled identically, pull scale system outside
      :NAME SCALED-INPUT
      :OBJECT (OUTPUT-OF ?SYSTEM@(NOT (OR SCALE-SYSTEM SHIFT-SYSTEM))
                    &REST ?INPUTS${(OUTPUT-OF ?SCALE ?[SCALE-INPUTS])})
                                      ; the inputs are all outputs from a single scale system.
                                      ; ?SCALE-INPUTS will be bound to the list of inputs to the scale system.
      :ANSWER (OUTPUT-OF SCALE (APPLY 'OUTPUT-OF SYSTEM SCALE-INPUTS)))
    (GOAL EQUIVALENT-FORM        ; if all the inputs are scaled, pull one of the scales outside
      :NAME UNEQUALLY-SCALED-INPUT
      :OBJECT (OUTPUT-OF ?SYSTEM@(NOT (OR SCALE-SYSTEM SHIFT-SYSTEM))
                    &REST ?INPUTS${(OUTPUT-OF (SPECIFIC-MEMBER SCALE-SYSTEM
                                                             ?[SCALE-FACTORS])
                                            ?[SCALE-INPUTS])})
                                         ; the inputs are all outputs from scale systems.
                                         ; ?SCALE-FACTORS will be bound to the list of the amounts of the scales.
                                         ; ?SCALE-INPUTS will be bound to the list of inputs to the scale system.
      :ANSWER
      (LET ((COMMON-SCALE (FIRST SCALE-FACTORS)))       ; pull the first scale outside
        (OUTPUT-OF (SCALE COMMON-SCALE)
          (APPLY 'OUTPUT-OF SYSTEM
                 (MAPCAR
                   '(LAMBDA (SCALE-FACTOR SCALE-INPUT)
                      (OUTPUT-OF (APPLY 'SCALE
                                              (MAPCAR '$-       ; compensate for the outside scale
                                                SCALE-FACTOR COMMON-SCALE))
                        SCALE-INPUT))
                   SCALE-FACTORS SCALE-INPUTS))))))
    ... ))
```

Figure 3-19: The definition for the system class, generalized-homogeneous-system

be highlighted. The representation and manipulation of abstract signals are supported: this support allowed the manipulation of the general modulated filter bank in which no specific choice had been made about the identity of the window.

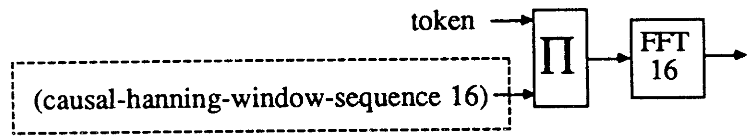
The general process by which alternate implementations are obtained was discussed. This involves not only finding identity transformations which are applicable to the given signal-processing expression but also finding transformations which are applicable to its subexpressions and to any of the newly uncovered equivalent implementations. The idea of regularity constraints was introduced as one way to limit the combinatorial growth which results from the independent manipulation of subexpressions. These regularity constraints are used to reduce the design space which is explored in the search for equivalent implementations.

The power of automatic algorithm manipulation was demonstrated using the modulated filter bank within the FSK-code detector. Three innovative implementations of the 16-channel, modulated filter bank were developed using ADE, the first using a 16-point rectangular window; the second, a 16-point Hanning window; and the third, a 32-point Hanning window. None of these three structures have been found in the published literature on modulated filter banks or short-time Fourier transforms. All three of these algorithms exhibit a high internal branching factor, making unconstrained manipulation of the algorithms untenable due to the number of possible subexpression combinations. Thus, this application area has shown that an algorithm design environment can have a marked beneficial effect on the solution of signal-processing problems and that, at least for this application, the use of regularity constraints is essential for solving these problems.

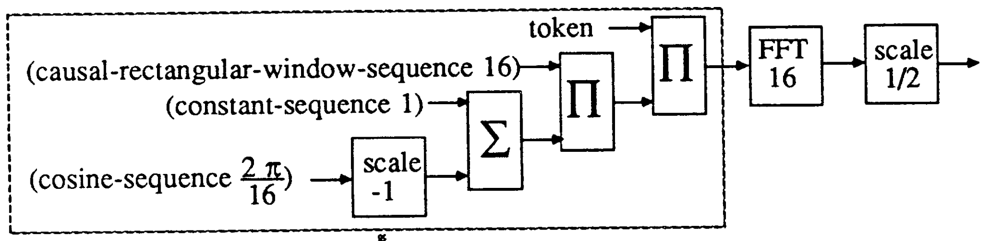
The remainder of this thesis explores issues involved in providing a design environment with capabilities such as those demonstrated here.

Addendum 3.A The sequence of transformations used in going from the FFT of the product involving the Hanning window to the sum of scaled and shifted versions of the FFT of the product of involving the rectangular window

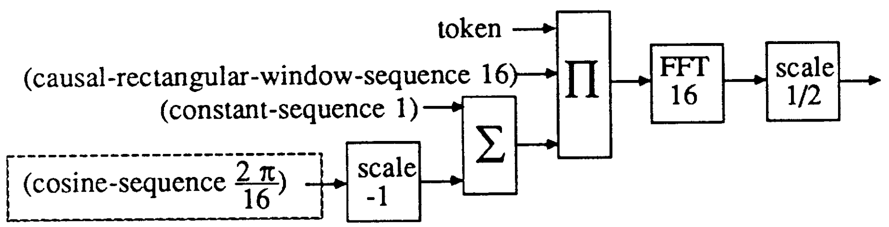
let "token" represent the abstract discrete-time sequence generated by the short-time Fourier transform output equivalent-form rule "using-1d-fft"



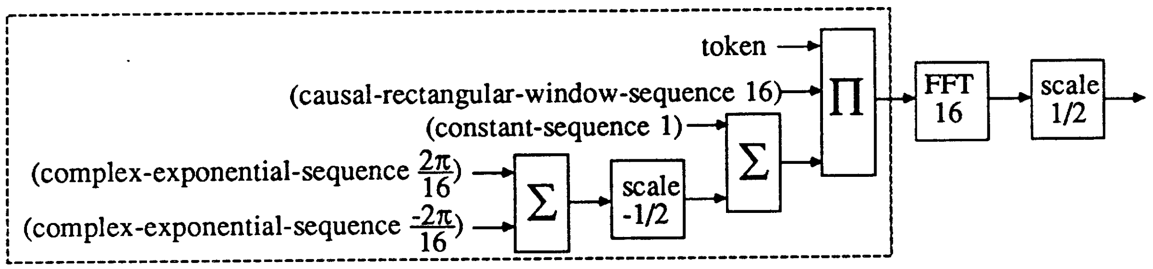
causal hanning-window sequence equivalent-form rule "master-copy" + simplification


token
(causal-rectangular-window-sequence 16)
(constant-sequence 1)
(cosine-sequence $\frac{2\pi}{16}$)
scale -1
Σ
Π
Π
FFT 16
scale 1/2

commutative, associative system output equivalent-form rule "self-application"

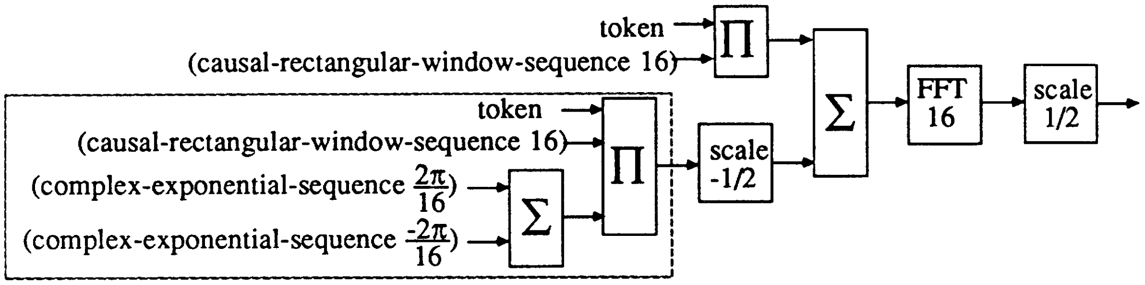

token
(causal-rectangular-window-sequence 16)
(constant-sequence 1)
(cosine-sequence $\frac{2\pi}{16}$)
scale -1
Σ
Π
FFT 16
scale 1/2

cosine sequence equivalent-form rule "master-copy" + simplification


token
(causal-rectangular-window-sequence 16)
(constant-sequence 1)
(complex-exponential-sequence $\frac{2\pi}{16}$)
(complex-exponential-sequence $\frac{-2\pi}{16}$)
Σ
scale -1/2
Σ
Π
FFT 16
scale 1/2

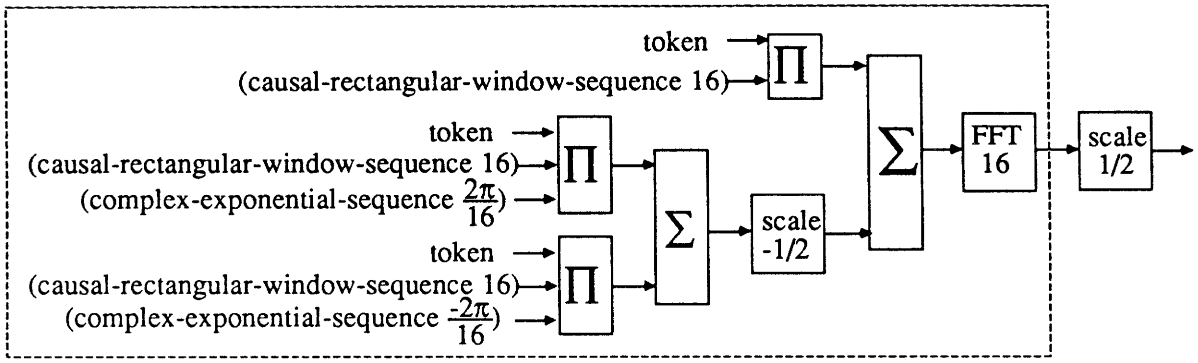
additive-system output equivalent-form rule "added-input" + simplification

additive-system output
equivalent-form rule "added-input"
+ simplification



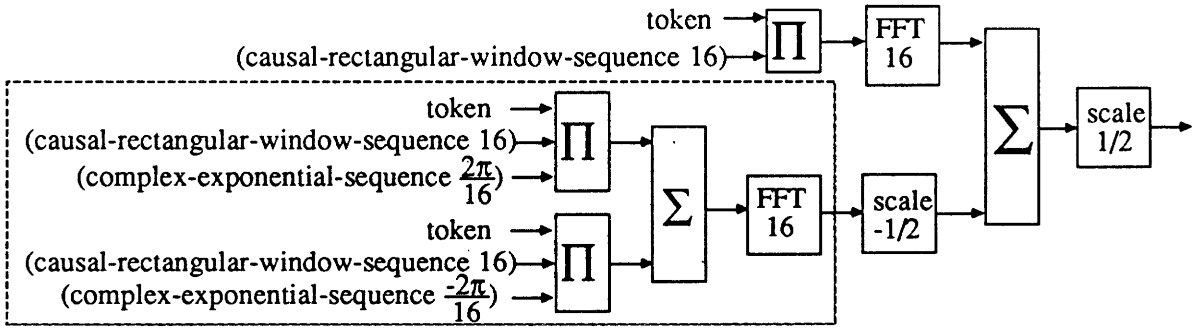
token
(causal-rectangular-window-sequence 16)
token
(causal-rectangular-window-sequence 16)
(complex-exponential-sequence $\frac{2\pi}{16}$)
(complex-exponential-sequence $\frac{-2\pi}{16}$)
Σ
Π
scale -1/2
FFT 16
scale 1/2

additive-system output
equivalent-form rule "added-input"



token
(causal-rectangular-window-sequence 16)
token
(causal-rectangular-window-sequence 16)
(complex-exponential-sequence $\frac{2\pi}{16}$)
token
(causal-rectangular-window-sequence 16)
(complex-exponential-sequence $\frac{-2\pi}{16}$)
Π
Σ
scale -1/2
FFT 16
scale 1/2

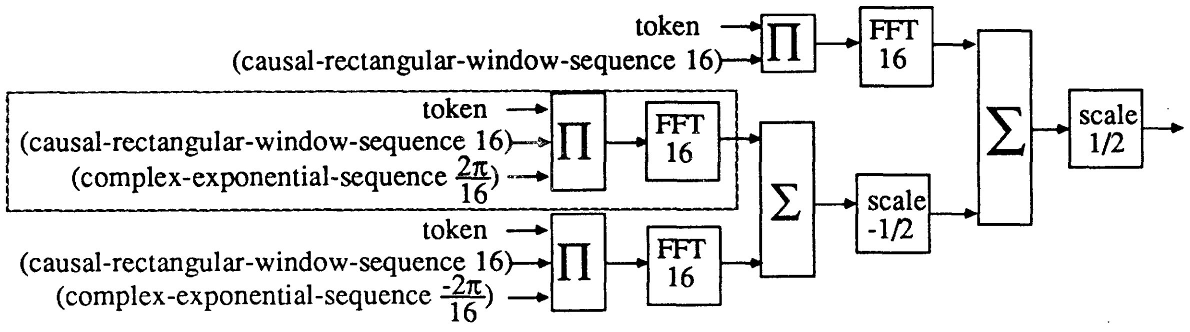
additive-system output
equivalent-form rule "added-input"
+ simplification



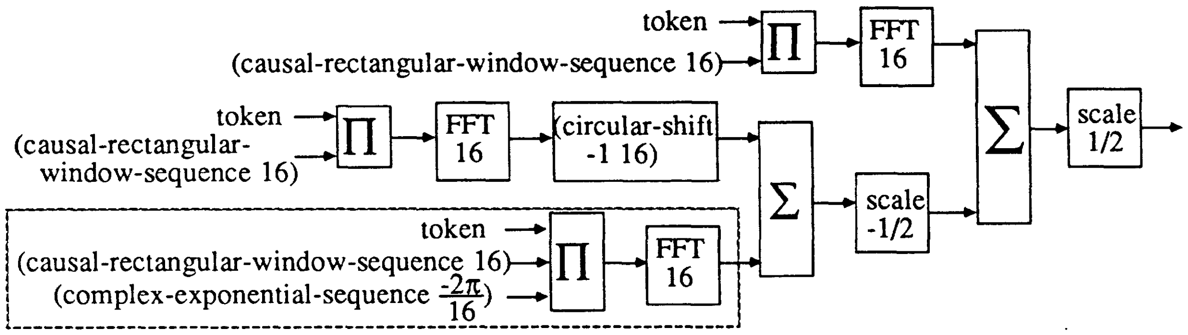
token
(causal-rectangular-window-sequence 16)
token
(causal-rectangular-window-sequence 16)
(complex-exponential-sequence $\frac{2\pi}{16}$)
token
(causal-rectangular-window-sequence 16)
(complex-exponential-sequence $\frac{-2\pi}{16}$)
Π
Σ
FFT 16
scale -1/2
scale 1/2

additive-system output
equivalent-form rule "added-input"

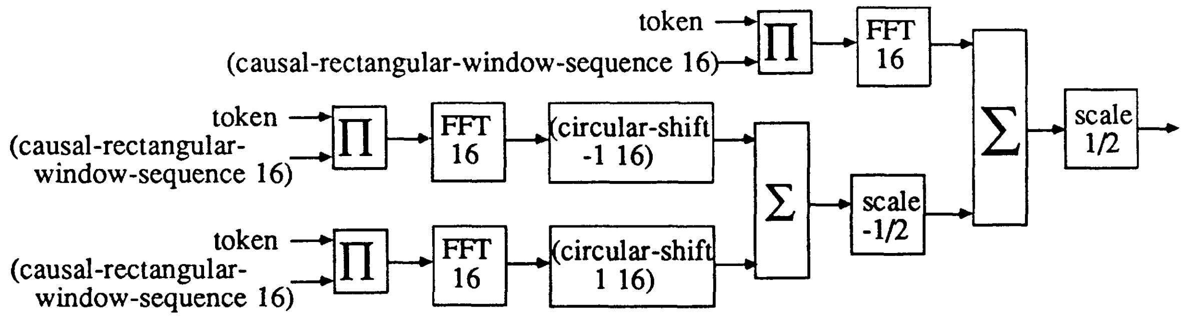
additive-system output
equivalent-form rule "added-input"


token
(causal-rectangular-window-sequence 16)
Π
FFT 16
token
(causal-rectangular-window-sequence 16)
(complex-exponential-sequence $\frac{2\pi}{16}$)
Π
FFT 16
token
(causal-rectangular-window-sequence 16)
(complex-exponential-sequence $\frac{-2\pi}{16}$)
Π
FFT 16
Σ
scale -1/2
Σ
scale 1/2

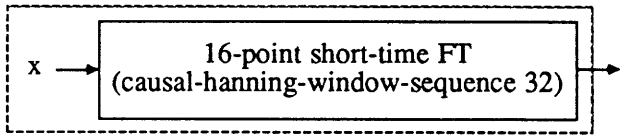
FFT output equivalent-form
rule "modulated-input"


token
(causal-rectangular-window-sequence 16)
Π
FFT 16
token
(causal-rectangular-window-sequence 16)
Π
FFT 16
(circular-shift -1 16)
token
(causal-rectangular-window-sequence 16)
(complex-exponential-sequence $\frac{-2\pi}{16}$)
Π
FFT 16
Σ
scale -1/2
Σ
scale 1/2

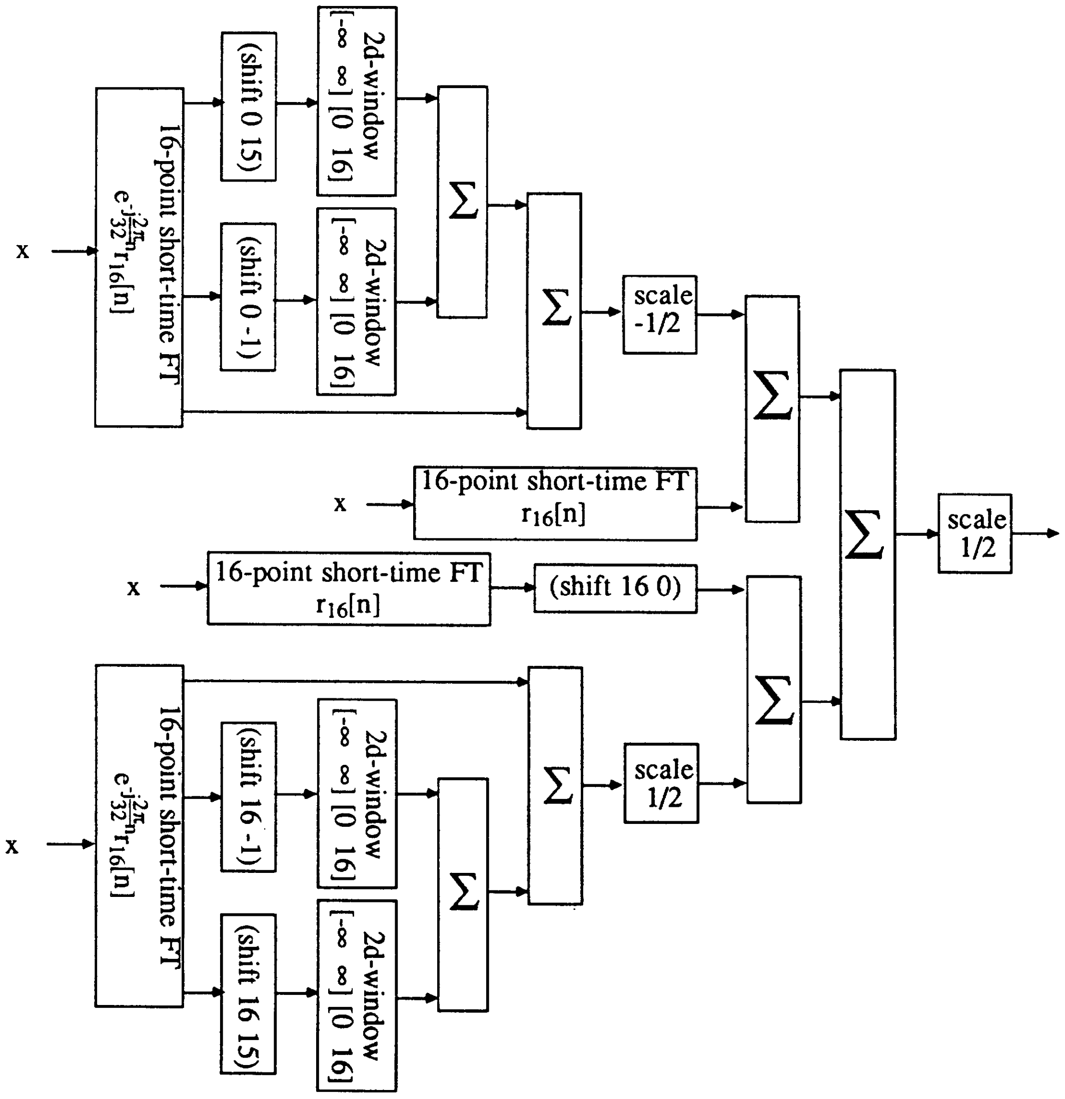
FFT output equivalent-form
rule "modulated-input"


token
(causal-rectangular-window-sequence 16)
Π
FFT 16
token
(causal-rectangular-window-sequence 16)
Π
FFT 16
(circular-shift -1 16)
token
(causal-rectangular-window-sequence 16)
Π
FFT 16
(circular-shift 1 16)
Σ
scale -1/2
Σ
scale 1/2

Addendum 3.B The sequence of transformations used in going from the 16-point short-time Fourier transform with a 32-point Hanning window to the structure shown in Figure 3-18

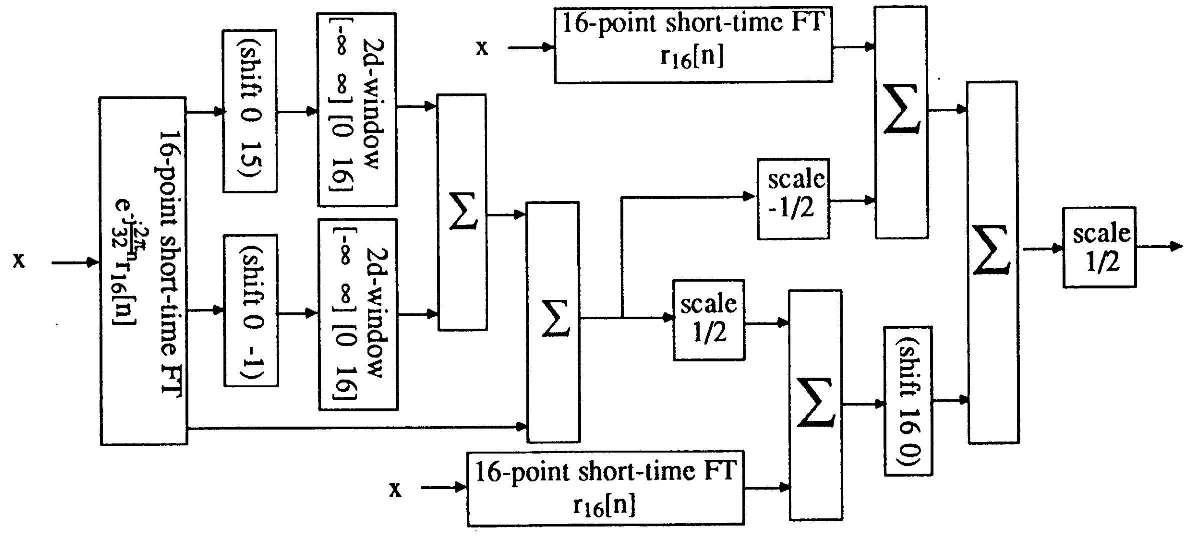


short-time FT output equivalent-form rule "using-1D-FFT"
(see development labelled "1D FFT transformations")
+ simplification



two applications of
2d-window output equivalent-form rule "shifted-input"
+ simplification

two applications of
2d-window output equivalent-form rule "shifted-input"
+ simplification



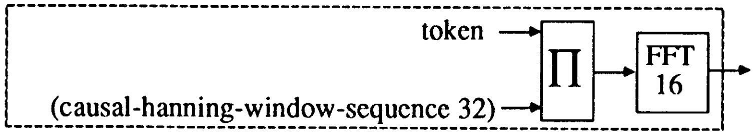
multiple applications of
generalized homogeneous system equivalent-form rule "unequally-scaled-input"
+ simplification

multiple applications of
generalized shift-invariant system equivalent-form rule "unequally-shifted-input"
+ simplification

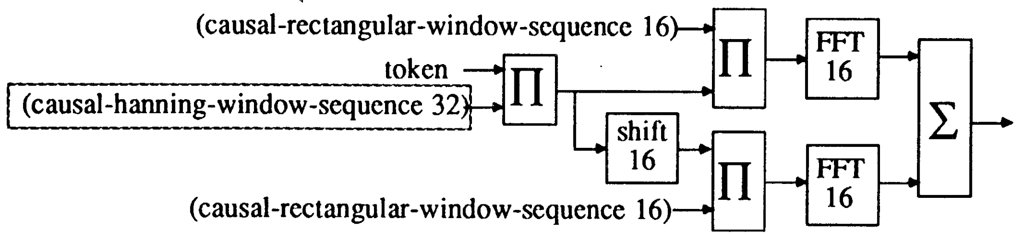
structure shown in Figure 3-18

1D FFT transformations

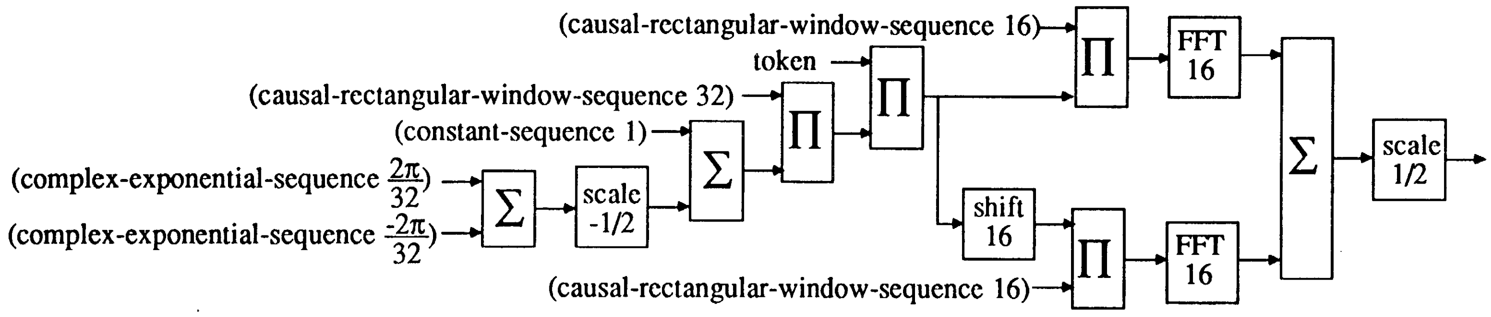
let "token" represent the abstract discrete-time sequence generated by the short-time Fourier transform output equivalent-form rule "using-1d-fft"



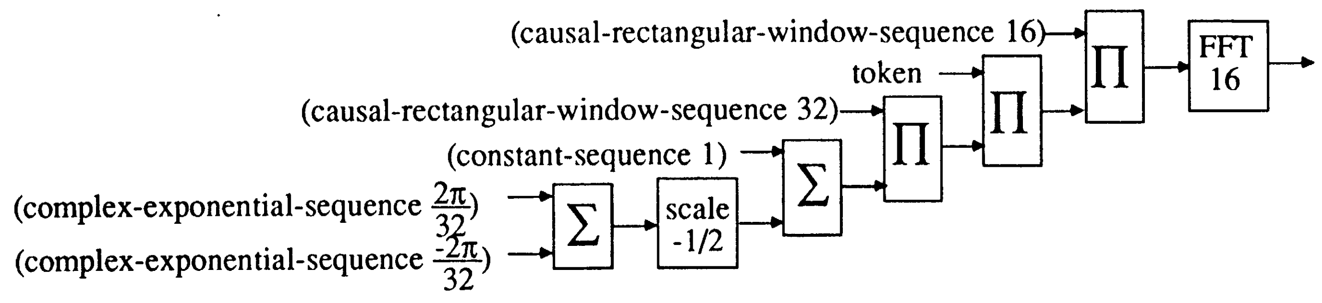
FFT output equivalent-form rule "aliased-input"


(causal-rectangular-window-sequence 16)
token
(causal-hanning-window-sequence 32)
Π
shift 16
(causal-rectangular-window-sequence 16)
Π
Π
FFT 16
FFT 16
Σ

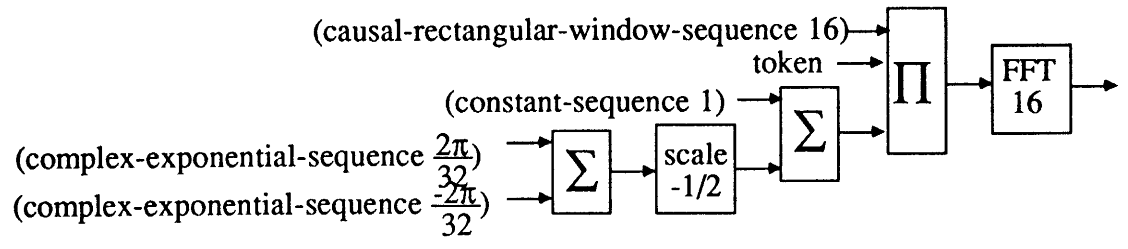
causal hanning-window sequence equivalent-form rule "master-copy"
+ simplification
cosine sequence equivalent-form rule "master-copy"
+ simplification


(causal-rectangular-window-sequence 16)
token
(causal-rectangular-window-sequence 32)
(constant-sequence 1)
(complex-exponential-sequence 2π/32)
(complex-exponential-sequence -2π/32)
Σ
scale -1/2
Σ
Π
Π
shift 16
(causal-rectangular-window-sequence 16)
Π
Π
FFT 16
FFT 16
Σ
scale 1/2

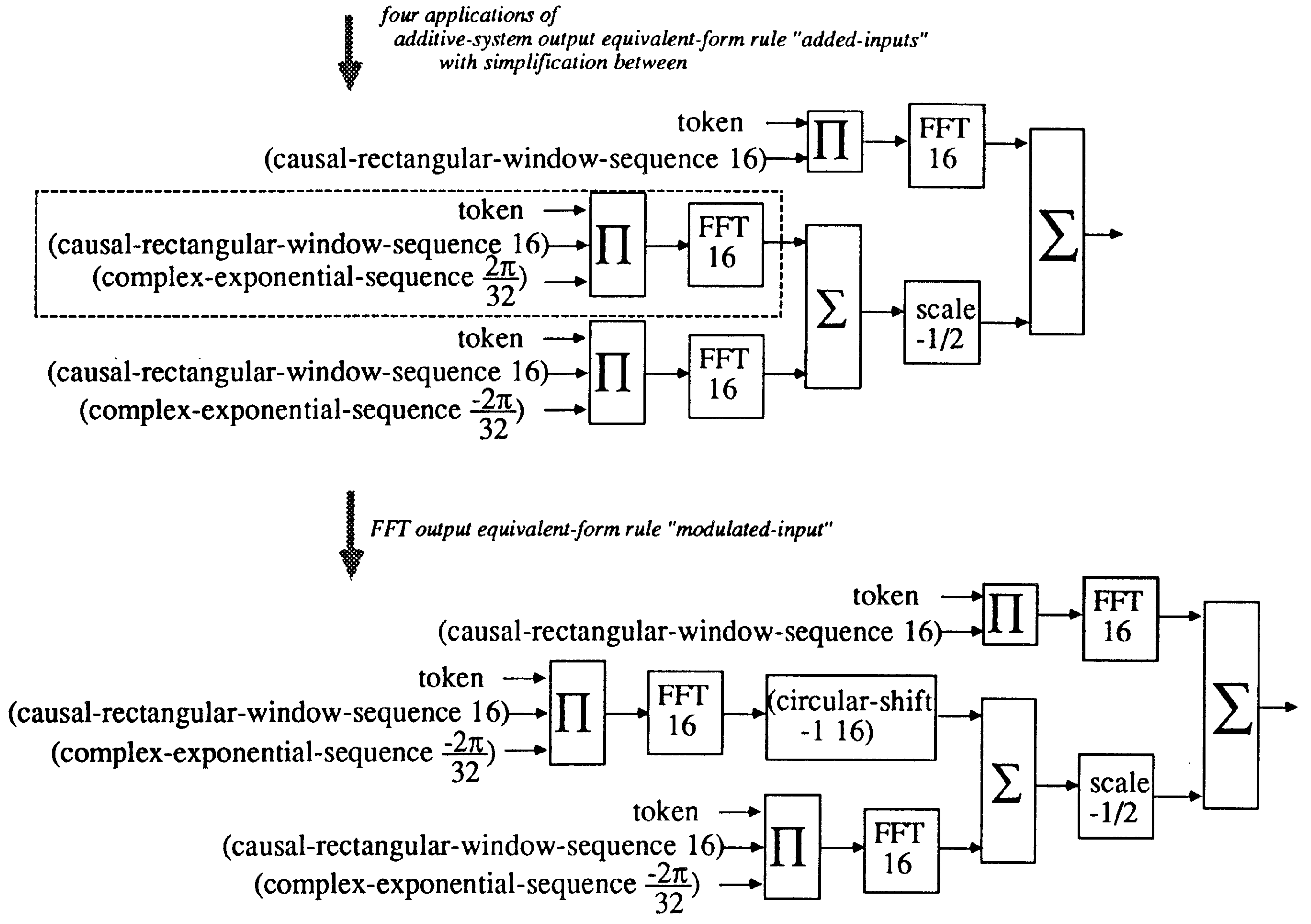
1D FFT transformations continued: manipulation of


(causal-rectangular-window-sequence 16)
token
(causal-rectangular-window-sequence 32)
(constant-sequence 1)
(complex-exponential-sequence $\frac{2\pi}{32}$)
(complex-exponential-sequence $\frac{-2\pi}{32}$)
Σ
scale -1/2
Σ
Π
Π
Π
FFT 16

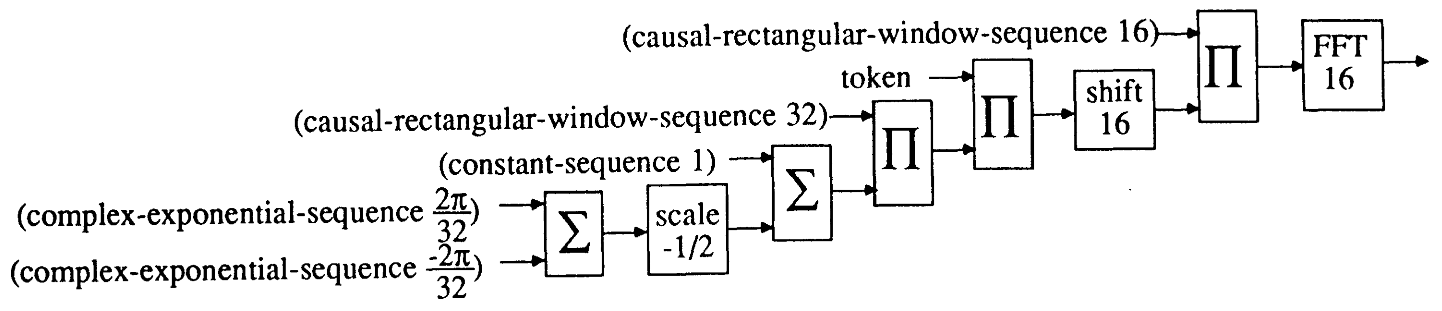
two applications of
commutative, associative system output equivalent-form rule "self-application"
+ simplification


(causal-rectangular-window-sequence 16)
token
(constant-sequence 1)
(complex-exponential-sequence $\frac{2\pi}{32}$)
(complex-exponential-sequence $\frac{-2\pi}{32}$)
Σ
scale -1/2
Σ
Π
FFT 16

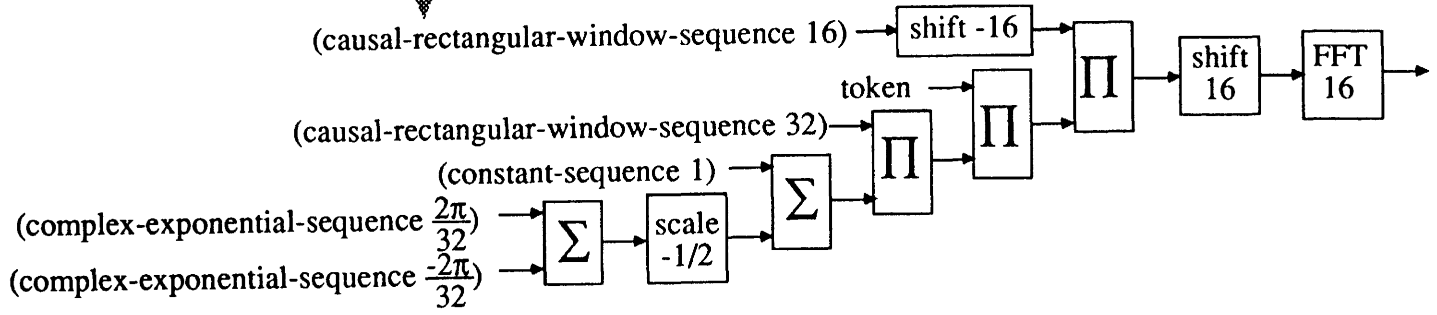
four applications of
additive-system output equivalent-form rule "added-inputs"
with simplification between


four applications of
additive-system output equivalent-form rule "added-inputs"
with simplification between
token
(causal-rectangular-window-sequence 16)
Π
FFT 16
Σ
(complex-exponential-sequence $\frac{2\pi}{32}$)
(complex-exponential-sequence $\frac{-2\pi}{32}$)
scale -1/2
FFT output equivalent-form rule "modulated-input"
(circular-shift -1 16)

1D FFT transformations continued: manipulation of

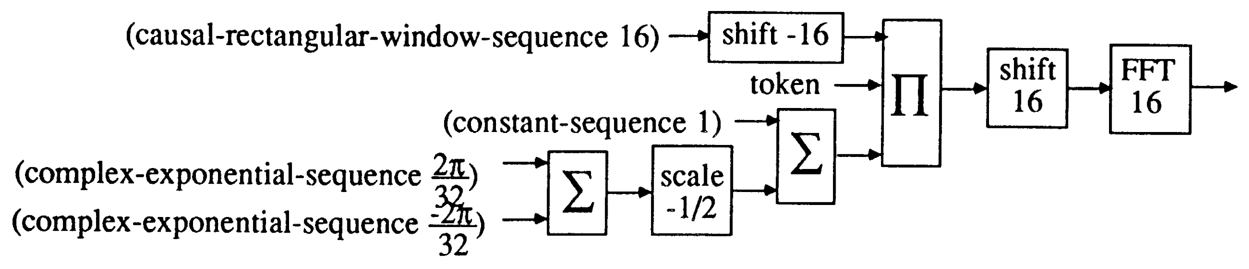

(causal-rectangular-window-sequence 16)
token
(causal-rectangular-window-sequence 32)
(constant-sequence 1)
(complex-exponential-sequence $\frac{2\pi}{32}$)
(complex-exponential-sequence $\frac{-2\pi}{32}$)
Σ
scale -1/2
Σ
Π
Π
shift 16
Π
FFT 16

generalized shift-invariant system output equivalent-form rule "single-shifted-input"

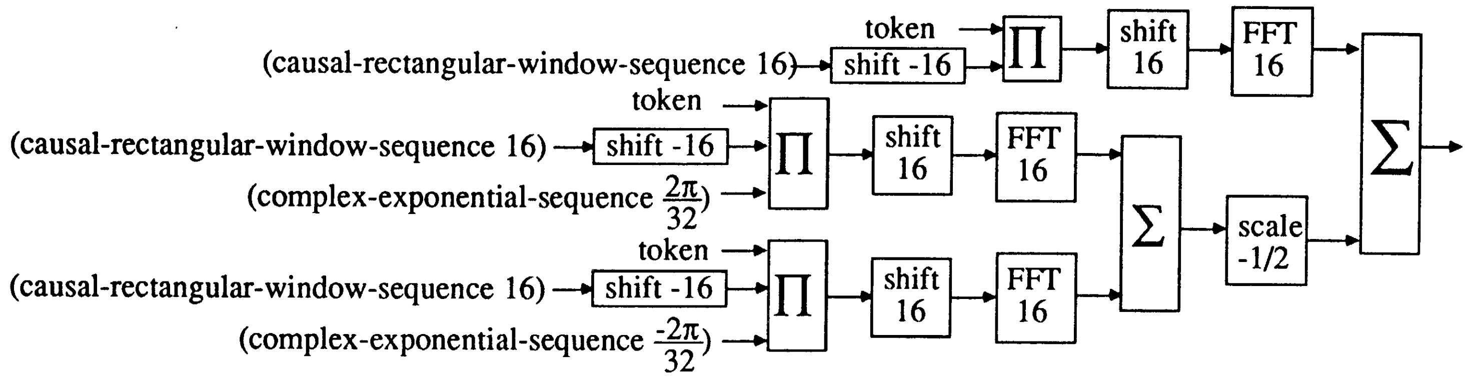

(causal-rectangular-window-sequence 16)
shift -16
token
(causal-rectangular-window-sequence 32)
(constant-sequence 1)
(complex-exponential-sequence $\frac{2\pi}{32}$)
(complex-exponential-sequence $\frac{-2\pi}{32}$)
Σ
scale -1/2
Σ
Π
Π
Π
shift 16
FFT 16

two applications of
commutative, associative system output equivalent-form rule "self-application"
+ simplification

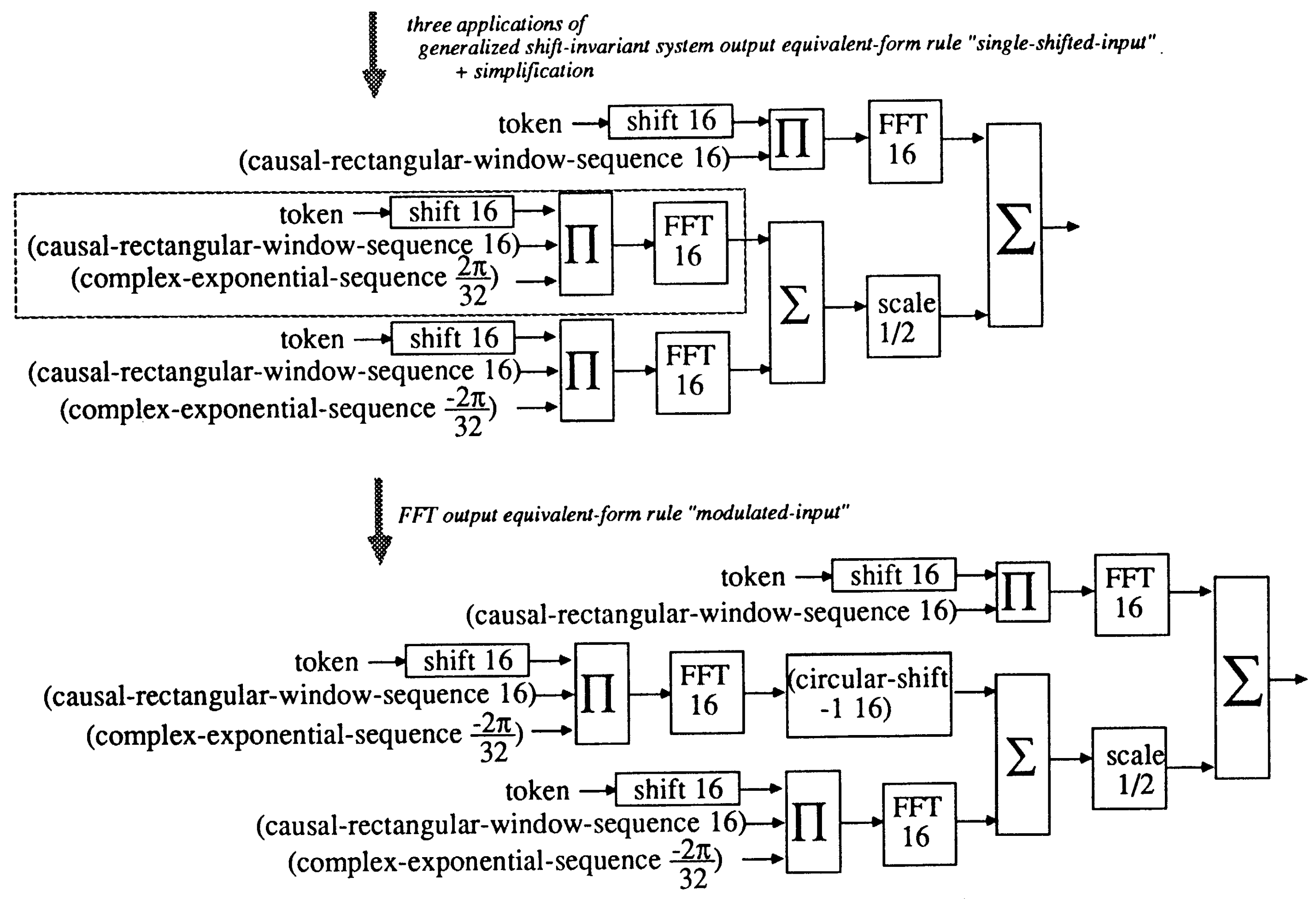
two applications of
commutative, associative system output equivalent-form rule "self-application"
+ simplification


(causal-rectangular-window-sequence 16)
shift -16
token
(constant-sequence 1)
(complex-exponential-sequence $\frac{2\pi}{32}$)
(complex-exponential-sequence $\frac{-2\pi}{32}$)
Σ
scale -1/2
Σ
Π
shift 16
FFT 16

two applications of
additive-system output equivalent-form rule "added-inputs"
+ two applications of
additive-system output equivalent-form rule "shifted-added-inputs"
with simplification between


token
(causal-rectangular-window-sequence 16)
shift -16
Π
shift 16
FFT 16
token
(causal-rectangular-window-sequence 16)
shift -16
(complex-exponential-sequence $\frac{2\pi}{32}$)
Π
shift 16
FFT 16
token
(causal-rectangular-window-sequence 16)
shift -16
(complex-exponential-sequence $\frac{-2\pi}{32}$)
Π
shift 16
FFT 16
Σ
scale -1/2
Σ

three applications of
generalized shift-invariant system output equivalent-form rule "single-shifted-input"
+ simplification


three applications of
generalized shift-invariant system output equivalent-form rule "single-shifted-input"
+ simplification
token
shift 16
(causal-rectangular-window-sequence 16)
(complex-exponential-sequence $\frac{2\pi}{32}$)
(complex-exponential-sequence $\frac{-2\pi}{32}$)
Π
FFT 16
Σ
scale 1/2
FFT output equivalent-form rule "modulated-input"
(circular-shift -1 16)

Chapter 4

Signal and System Representation

Chapter 1 described the central goal of this thesis, namely the exploration of automatic signal-processing expression manipulation. The concepts central to this research are explored through the definition and implementation of an environment embodying these concepts. Areas involved in describing such an environment can be divided into the representation of signals and systems and the behavior of the control structures used to manipulate them and their properties. This chapter and the following chapter discuss these issues: this chapter explores the characteristics needed in complete signal and system representations and the next chapter explores the characteristics of an appropriate control structure.

4.1 Signal and System Representation in the Algorithm Design Environment (ADE)

The Algorithm Design Environment, described in Chapter 3, is used to demonstrate the validity and the power of the ideas presented in this thesis. This section provides a more detailed description of the signal and system representations used in ADE.

Table 4.1: Categories of signals and systems supported in ADE

Domain	Information content	Non-zero support	Computability
Discrete-time domain	Simple specific	Finite-length	Computable
Discrete-time,	Symbolically constrained	Left-sided	Uncomputable
Fourier-transform domain	Abstract	Right-sided	
Z-transform domain		Doubly infinite	

4.1.1 Signal and system manipulation

As shown in Table 4.1, many categories of signals and systems are represented in ADE. Most of these categories are self-explanatory. The only categorization which will be discussed in detail here is that of information content.

Simple, specific signals and systems are completely specified objects. Examples of simple, specific signals include $e^{-j\frac{\pi}{8}3n}$ and $r_{16}[n] * r_{16}[n]$ where $r_{16}[n]$ is the 16-point, causal rectangular window. The objects are completely specified and all property values and sample values can be explicitly described.

As discussed in Chapter 3, abstract signals and systems represent an incomplete description of the signal or system. Examples of abstract signals and systems include objects described solely by the signal class to which they belong, like #<(A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE)> or #<(A-MEMBER-OF 'LINEAR-SYSTEM)>. Descriptions of abstract signals and systems can also include additional information, as illustrated on line *I-2* of Figure 3-4. The difference between abstract and specific signals is that the description given for the abstract signal can refer to any of a multitude of distinct signals whereas the description given for the specific signal refers to a single, unique signal. The same distinction holds for abstract and specific systems.

Symbolically constrained signals and systems are best described by example: the output from the FSK-code detectors on line *O-4* of Figure 3-4 is a symbolically constrained signal as are the sequences manipulated on lines *I-7* and *I-8*. They are not simple, specific signals, in that no simple mapping from their indices to their sample values can be given. On the other hand, symbolically constrained signals are specific signals, not abstract signals: their specification refers to a single, unique signal. Thus, they lie between simple, specific signals and abstract signals in their information content: their identity can be uniquely determined but no simple complete description can be given for their sample values.

To simplify the user's task, a uniform interface is maintained that is as independent of the identity of the particular signal or system as possible. As in SPLICE (Dove et al., 1984; Myers, 1986), access to sample values is independent of the programming paradigm used in defining the signal. Similarly, this access is independent of the domain of the signal: sample-value retrieval from a discrete-time Fourier-transform signal or from a z-transform signal is the same as from a discrete-time sequence. Furthermore, any sample value can be retrieved, even if the signal is abstract or the actual, numeric value is uncomputable. In these two cases, a "symbolic number" is used as an answer, in the same way as $x[5]$ is used mathematically as the sample value for an abstract, discrete-time sequence, $x[n]$. Constraints are imposed on the type and the magnitude of the symbolic number based on the sample type and range of the signal.

All signals and systems are *apparently* immutable, in that their observable property and sample values are inalterable: the representation of the signal or system can not be altered in an inconsistent manner by subsequent processing.

Referring to a signal or system returns its representation, without the overhead involved in determining all the property and sample values. Instead, determination of each of these quantities is deferred until the value is needed. Once a property or sample value is determined, it is recorded so that subsequent references need not recompute the value.

Finally, each unique signal and system has a unique representation. If multiple objects were allowed to represent a unique signal or system, the redundant representations would consume additional resources.

4.1.2 Signal and system definition

Some of the signal and system characteristics which simplify their definition coincide with those already listed for their external manipulation. In particular, immutability, unique representation, deferred evaluation and caching of property and sample values are all behaviors which simplify the definition of signals and systems.

As was illustrated in Figure 3-3, signal-class definitions are collections of information about the property and sample values of the signals. Similarly, system class definitions are collections of information about the property values of the systems and the property and sample values of their output signals. Separate descriptions of sample values can be associated with each signal or system class. Another level of modularity is also available for property value descriptions: these descriptions tend to rely on the details of the signal or system as well as its class. For example, the Fourier transform of a product is described using multiple independent forms (lines 20–27 and lines 28–40 of Figure 4-1). Similarly, the description of the simplification of a product is given by multiple partial descriptions, some of which are included in the definition of SEQUENCE-MULTIPLY (lines 7–10 and lines 11–19 of Figure 4-1) and some of which are included in the definitions of its superclasses.

The description of sample values is a major component of signal- and system-class definitions. To simplify this programming task, modularity and a variety of programming paradigms are supported. As in SPLICE (Dove et al., 1984; Myers, 1986), array-based models, point-based models, state-machine and composition models are all supported for describing sample values. The non-zero support is used to restrict the sample-value requests to within this support: this can often be used to omit explicit bounds checking from the sample-value functions. Similarly, the periodicity of a periodic signal is used to shift all sample-value requests down to the period extending upward from zero, potentially reducing the time and space required to compute the requested sample values.

```
(DEFINE-SYSTEM SEQUENCE-MULTIPLY (&REST INPUTS@(LIST-OF DISCRETE-TIME-SEQUENCE))
                                    ; accept any number of discrete-time sequences as inputs
    (MULTIPLY-SYSTEM DISCRETE-TIME-SYSTEM)        ; a subclass of these classes
  ("the system for multiplying sequences")
  NIL ()                            ; generate a new output signal class, without any additional superior signal classes
  ("the product of sequences"
    (GOAL SIMPLIFICATION         ; if there are no inputs, the "product" is a unity-gain constant sequence
      :NAME NO-INPUTS
      :OBJECT (OUTPUT-OF ?MULTIPLY)
      :ANSWER (CONSTANT-SEQUENCE 1))
    (GOAL SIMPLIFICATION         ; if all the inputs are IZT's, use the z-domain representation for the multiply
      :NAME IZT-INPUTS
      :OBJECT (OUTPUT-OF ?MULTIPLY &REST
                  ?INPUTS${(OUTPUT-OF (SPECIFIC-MEMBER IZT-SYSTEM) ?[ZT-SIGNALS])})
                                    ; the inputs are all outputs from an inverse z transform.
                                    ; ?ZT-SIGNALS will be bound to the list of inputs to the inverse z transforms.
      :ANSWER (INVERSE-Z-TRANSFORM
                  (OUTPUT-OF (ZT-SCALE (/ 1 (MAX 1 (* (1- (LENGTH INPUTS)) 2PI))))
                    (APPLY 'ZT-CONVOLVE Z-TRANSFORMS))))
    (GOAL FT                        ; if all the Fourier transforms for all the inputs exist,
                                    ; the Fourier transform of the product is the convolution of the transforms of the inputs
      :NAME AS-CONVOLUTION-INPUT-FT
      :OBJECT ?SELF              ; any product.
      :WHEN (NEVER '(LAMBDA (INPUT) (SAMEP (FOURIER-TRANSFORM INPUT) NONE))
                       INPUTS) ; all the Fourier transforms of the inputs exist
      :ANSWER (OUTPUT-OF (SIGNAL-SCALE (/ 1 (MAX 1 (* (1- (LENGTH INPUTS)) 2PI))))
                   (APPLY 'SIGNAL-CONVOLVE (MAPCAR 'FOURIER-TRANSFORM INPUTS))))
    (GOAL FT                        ; if the product can be expressed as a complex exponential times a subproduct
                                    ; and the Fourier transform of the subproduct exists, the Fourier
                                    ; transform of the product is a shifted version of the Fourier transform of the subproduct
      :NAME COMPLEX-EXPONENTIAL-MODULATED-INPUT
      :OBJECT (OUTPUT-OF ?MULTIPLY
                  (SPECIFIC-MEMBER COMPLEX-EXPONENTIAL-SEQUENCE ?FREQ)
                  &REST ?OTHER-INPUTS)
                                    ; one of the inputs is a complex-exponential sequence
      :WHEN (NOT (SAMEP (FOURIER-TRANSFORM
                               (APPLY 'SEQUENCE-MULTIPLY OTHER-INPUTS))
                             NONE))       ; the Fourier transform of the subproduct exists
      :ANSWER (OUTPUT-OF (SIGNAL-SHIFT ($MINUS FREQ))
                   (FOURIER-TRANSFORM (APPLY 'SEQUENCE-MULTIPLY OTHER-INPUTS))))
    ... ))
```

Figure 4-1: An example of a system-class definition

4.1.3 Summary

This section has briefly discussed some of the characteristics which are important in signal and system representations. Some of these characteristics will be considered in this chapter and the next. Others, such as apparent immutability and unique representation, are discussed by Kopec (1980), Dove et al. (1984) and Myers (1986). The remainder of this chapter discusses the definition of hierarchies of signal and system classes. The basic signal and system representations are also briefly considered. Finally, new representations for abstract objects and for some symbolically constrained objects are developed.

4.2 Representational Hierarchies for Signals, Systems and their Classes

Since the primary focus of signal processing is the systems and signals involved, both signals and systems are commonly dealt with and thought of as express entities. A simple way to support this conceptualization is to represent signals and systems using object-oriented programming. Since the signal and system representations in ADE rely on object-oriented programming, the following subsection is devoted to the description of this programming paradigm. Subsection 4.2.2 then describes the actual representations used in ADE.

4.2.1 Object-oriented programming

The focus of object-oriented programming, appropriately, is objects. Objects are used to combine the properties of procedures and data by both performing computations and saving local state. Message passing between objects is used as an indirect procedure call: the object receiving a message selects a method from its private procedures for completing the indicated operation. The close association between the data objects and the procedures supports both data abstraction and polymorphism, both essential prerequisites to program modularity.[1]

Constructs in object-oriented programming fall in two major categories: classes and instances. A class describes similar objects and instances are the individual objects in the class. For example, using object-oriented programming, "number" would be a class and "$\frac{3}{5} + j\frac{4}{5}$" would be an instance of the type "number". The class descriptions enumerate the instance variables, the variables included in each instance, and define methods, the local procedures used by the instances to respond to messages. Continuing the example using numbers, "real-part" and "imag-part" would be two instance variables of objects within the class "number" and "magnitude" and "phase" would be two methods available to these objects.

Methods and instance variables can be provided either directly by the containing class or indirectly by inheritance between classes. By defining hierarchies of classes, the instances of a subclass will inherit the instance variables and the methods defined for the superclass. This allows independent aspects of an object to be described separately and allows similarities between classes to be made explicit. Again using numbers, "real-number" and "imaginary-number" both would be subclasses of "number". Through inheritance of instance variables and methods, real and imaginary numbers would contain the instance variables, "real-part" and "imag-part" and would respond to the messages, "magnitude" and "phase".

When multiple methods for a single message are available to a class through inheritance and local definition, then the method from a subclass will take precedence over any corresponding method from any of its superclasses. Thus, another method could be defined for real numbers, giving the phase as 0 or π, according to the sign of the real number. This new method for phase would be used by all real numbers. In ADE, the class hierarchy is actually a lattice, with the possibility of multiple direct superclasses for a single class: for example, SEQUENCE-MULTIPLY is a direct subclass of both MULTIPLY-SYSTEM and DISCRETE-TIME-SYSTEM. When there are multiple direct superclasses, multi-

[1]Polymorphism is the use of a single function name to index multiple procedures. It can be used to emphasize the general unified purpose of the component procedures.

ple methods from these superclasses may still be inherited for a single message. Stefik and Bobrow (1986) provide a detailed description of alternate approaches used to resolve these contentions.

In summary, object-oriented programming provides a uniform interface for computing and saving local state. Multiple methods can be defined using a single name, each method being valid for a different class of objects. Class hierarchies and inheritance emphasize similarities between object classes. These properties make object-oriented programming an ideal basis on which to build signal and system representations.

4.2.2 Hierarchical organization of signal and system representations

The ability of object-oriented programming to branch according to the type of the signal or system provides modularity between classes. Thus, object-oriented methods are used to describe sample-value functions. Signal and system properties require greater modularity within their definitions. This requirement was illustrated in the previous section using the determination of the Fourier transform of a product of signals (Figure 4-1). This requirement for additional modularity is met using a rule-based representations: property information is given in the form of rules which use pattern matching to determine applicability. This approach will be discussed further in Chapter 5. While modularity can be provided using object-oriented programming, some of the other representational requirements described in section 4.1 can not be met without some extensions: in particular, unique representation, deferred evaluation and caching are not provided by this paradigm. These extensions as well as the underlying use of object-oriented programming are the subject of this subsection.

By recording the representations as they are created, unique representations of signals and systems can be achieved. In particular, using this approach, a unique representation of an output signal can be derived from the generating system and its inputs: the generating system would provide the caching table of signal objects and the list of inputs would provide a key. For example, for the sequence $y[n] = x_1[n]x_2[n]x_3[n]$, the system MULTIPLY would provide the table and the list, $(x_1[n] \quad x_2[n] \quad x_3[n])$ would act as the key. The first time an output signal is referenced, no entry would be found and a new signal object would be created and cached. All subsequent applications of that system to those inputs return the same signal representation, simply by looking it up in the caching table. A similar approach is taken for maintaining unique representations of systems and inherent signals.

The example given above, namely $y[n] = x_1[n]x_2[n]x_3[n]$, raises an interesting issue: to illustrate, consider the sequence $x_2[n]x_3[n]x_1[n]$. Given the above description, a distinct sequence will be created to represent $x_2[n]x_3[n]x_1[n]$. Yet, due to the commutativity and associativity of multiplication, the distinction between these forms is blurred. The generating systems, their inputs and, in fact, all the property and sample values of $x_1[n]x_2[n]x_3[n]$ will be identical to those of $x_2[n]x_3[n]x_1[n]$. Thus, these signals are essentially identical. Maintaining separate representations for them has all the inefficiencies of duplicating representations, both in repeated computation and in wasted space. To avoid these inefficiencies, a single representation can be used for all permutations of the inputs into commutative and associative systems. By reordering the caching keys canonically, this single representation will be accessed, independent of the given permutation of the inputs.

A similar issue arises in the representation of systems and of inherent signals. To illustrate, consider the complex-exponential sequences $e^{-j\frac{\pi}{8}n}$ and $e^{j\frac{\pi}{8}15n}$, retrieved from COMPLEX-EXPONENTIAL-SEQUENCE by the parameter values $-\frac{\pi}{8}$ and $\frac{15\pi}{8}$, respectively. The sequences are completely identical to one another and to any complex-exponential sequence of the form $e^{-j\frac{\pi}{8}(16k+1)n}$: they all belong to the same signal class and their sample and property values are all identical. A unique representation is possible if provisions are made for mapping the parameters to a canonical set of values. This mapping is included in the signal and system class definitions where multiple, distinct descriptions of a single signal or system are possible. Then, when a signal or system is required, the parameters and thus the indexing key are mapped to a canonical value before any attempt is made to retrieve the actual signal or system.

4.3 Abstract Objects and Specific Objects with Dependencies on Abstract Objects

The discussion up to this point has implied that all signal and system representations are cached and reused by all subsequent references. This behavior is not appropriate for abstract signals and systems: for example, two separate references to abstract discrete-time sequences should not be forced to refer to the same abstract discrete-time sequence. Otherwise, one could not talk about two distinct sequences, $x_1[n]$ and $x_2[n]$, both characterized by the description (A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE).

One option in representing abstract objects is to avoid caching any information and return completely separate objects for each invocation of an abstract description: thus, the objects returned by two separate invocations of (A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE) would be completely separate and unrelated. This is certainly a valid approach to the representation of abstract objects and it is the approach used in E-SPLICE (Myers, 1986). However, upon consideration, the abstract objects which share a common description are not completely unrelated: all the information which is known about them and all the information which is derivable about them is identical. For example, two distinct instances of an abstract discrete-time sequence obviously have the same signal domain. Since they share the same known properties and since they use the same rules to determine the remaining property values, they will also share the same non-zero support, sample type, bandwidth, symmetry characteristics and periodicity. More importantly, if a system $H\{\}$ is separately applied to two related abstract signals, x_1 and x_2, the output signals $H\{x_1\}$ and $H\{x_2\}$ are also related.[2] Since $H\{\}$ is unchanged between the

[2]This discussion assumes that the system $H\{\}$ is independent of $x_1[n]$, $x_2[n]$ and all other abstract objects.

two applications and since x_1 and x_2 are indistinguishable in terms of property values, $H\{x_1\}$ and $H\{x_2\}$ will also be indistinguishable in terms of property values: that is, $H\{x_1\}$ and $H\{x_2\}$ will be distinct but they will share the same non-zero support, sample type, bandwidth, symmetry characteristics and periodicity. Their simplifications and equivalent forms will also be closely related: in particular, replacing x_1 with x_2 in the simplification of $H\{x_1\}$ will give the simplification of $H\{x_2\}$ and the same replacement in the equivalent forms of $H\{x_1\}$ will give the equivalent forms of $H\{x_2\}$. When $H\{\}$ is a complicated signal-processing algorithm, such as that used for FSK-code detection, the computational resources used in determining the property values of the output is significant. It was this sharing of information which allowed the periodicity of the signal shown on line *I-8* of Figure 3-4 to be determined without any additional calculation. The signal on line *I-8* is related to the output from the modulated filter bank, used on line *I-7* of Figure 3-4. As shown in the annotations between lines *I-7* and *O-7*, the periodicity of this earlier modulated filter-bank output has already been determined. Since the modulated filter bank in both applications is the same and since the abstract instances to which it is applied are related, the information determined about the first output signal is reused to answer the question about the second. As will be discussed later in this section, the use of the one-dimensional Fourier transform to search for equivalent forms of the short-time Fourier transform provides another example of information sharing. The remainder of this section develops a representation for abstract objects which expedites the sharing of information between related instances.

4.3.1 Representation of abstract objects

As mentioned above, distinct representations must be returned for each invocation of an abstract description but the underlying information about the objects is either identical or closely related. This suggests that, although the outermost shells which are returned must be distinct, the underlying representation of the abstract description can be unique. Figure 4-2 illustrates this idea pictorially. The outermost shell of the

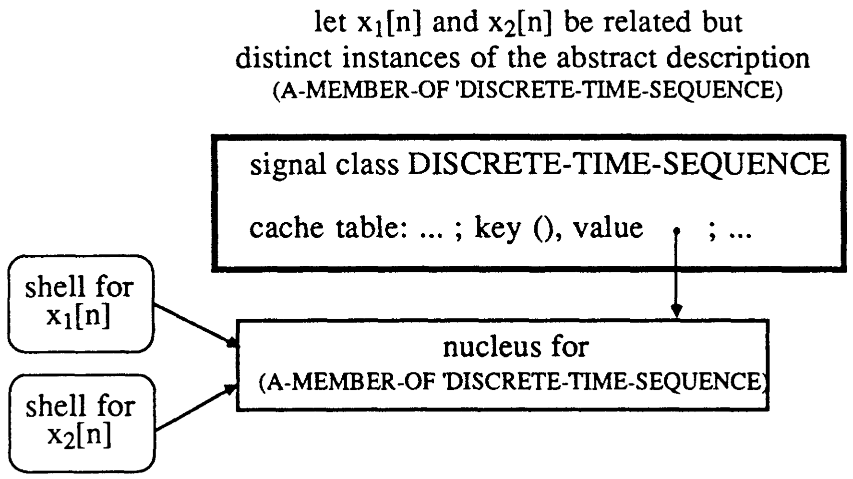

let $x_1[n]$ and $x_2[n]$ be related but
distinct instances of the abstract description
(A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE)
signal class DISCRETE-TIME-SEQUENCE
cache table: ... ; key (), value ; ...
shell for $x_1[n]$
shell for $x_2[n]$
nucleus for
(A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE)

Figure 4-2: Two-level representation for abstract signals and systems

Distinct representations must be returned for each invocation of an abstract description but the underlying information about the related abstract objects is either identical or closely related. One way to reflect this relationship between abstract instances is to use a two-level representation for abstract objects. The outermost shell of the representation forms a thin representational layer covering the nucleus of the abstract object. The nucleus of the abstract object in many ways behaves like a specific object: it is a unique representation for a particular signal or system description which is returned whenever that description is invoked. This combination allows related abstract objects to be distinguished by the shell object while still sharing a unique, central representation.

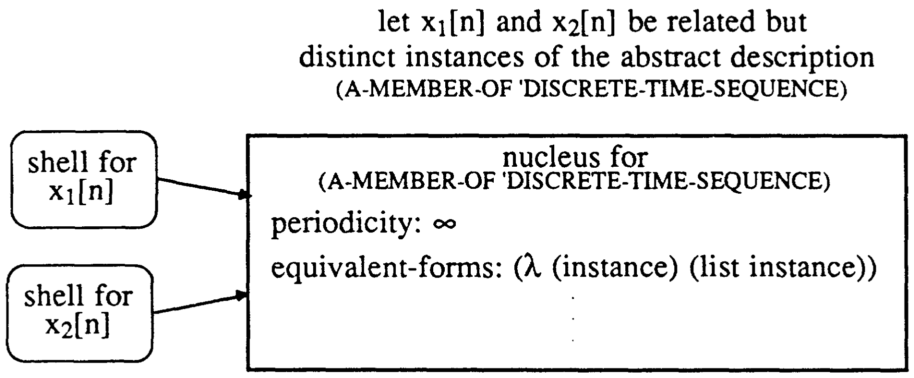

let $x_1[n]$ and $x_2[n]$ be related but
distinct instances of the abstract description
(A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE)
shell for $x_1[n]$
shell for $x_2[n]$
nucleus for
(A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE)
periodicity: ∞
equivalent-forms: (λ (instance) (list instance))

Figure 4-3: Centralization of knowledge determined about abstract objects

The property values of an abstract object are collected, after some modification, in the nucleus of the representation. Property values which depend on the identity of the owner are recorded in a functional form, with the owner as an argument. When an attempt is made to return one of these modified property values, the result of applying the functional description to the particular instance under consideration is returned instead.

representation will be referred to as the abstract shell, since it is simply used as a thin representational layer covering the nucleus of the abstract object. The nucleus of the abstract object in many ways behaves like a specific object: it is a unique representation for a particular signal or system description which is returned whenever that description is invoked. The difference lies in the fact that the abstract nucleus is never returned without a covering shell and a different shell object is generated on each invocation of the abstract description. This combination allows related abstract objects to be distinguished by the shell object while still sharing a unique, central representation of the information which is known or can be determined.

From the description of the abstract shells and the abstract nucleus, the shells need not be cached but the nucleus must be. Shell objects do not need to be cached, since they are never reused: a new shell is generated on each invocation of the abstract description. On the other hand, the abstract nucleus is expected to be shared between all the shells and thus is cached in order to allow its reuse.

The property values of an abstract object are collected in the nucleus of the representation. Modification is necessary on some property values. To illustrate, consider the determination of the equivalent forms of some abstract discrete-time sequence, $x_1[n]$. No alternate representations will be found for $x_1[n]$, so the equivalent forms of $x_1[n]$ will be the list $(x_1[n])$. This answer could be recorded directly in the nucleus but then the information in the nucleus would end up being a conglomerate, some parts referring to one instance of the abstract description and others, to another. Subsequent usage of the information would need to determine the modifications necessary to give, for example, the equivalent forms of $x_2[n]$. A simple and attractive alternative is to complete this examination once, when the property value is being recorded in the abstract nucleus. If the property value does not include the identity of the owner, then the value is recorded without modification. However, if the property value does include the identity of the owner, as the equivalent forms did above, then the property is flagged and a functional description of the property value recorded, with the abstract object as the argument (Figure 4-3). This simplifies the task of returning property values: either the property value is as recorded or its functional description will return the correct value when applied to the particular instance under consideration.

Unlike property values, the records of sample values of abstract signals should not be centralized. Two simultaneous constraints prevent the centralization of the sample-value records. The first is that two distinct instances of an abstract signal description should not be forced to have identical corresponding sample values: that is, if $x_1[n]$ and $x_2[n]$ are two abstract discrete-time sequences, there is no reason to believe that $x_1[3] = x_2[3]$ or that any of the other sample values are identical. So, distinct symbolic numbers must be used to represent the sample values of the distinct abstract instances. The second constraint is that a unique sample value must be given for repeated requests to a particular abstract signal: $x_1[3]$ should return a unique value, no matter how many times the request is made. Thus, each abstract shell must maintain its own records of its sample values. The same arguments apply to abstract systems and their output signals: for example, for two related abstract discrete-time systems, $H_1\{\}$ and $H_2\{\}$, $H_1\{\delta[n]\}$ is distinct from $H_2\{\delta[n]\}$ and $H_1\{\delta[n]\}$ should return a single unique signal no matter how many times the description is invoked. Thus, the records of the output signals from abstract systems must be maintained separately in each abstract system shell.

4.3.2 Representation of specific objects with dependencies on an abstract object

As mentioned at the beginning of this section, if a simple, specific system $H\{\}$ is separately applied to two related abstract signals x_1 and x_2, the output signals $H\{x_1\}$ and $H\{x_2\}$ are also easily and accurately related. In particular, since all the available information about x_1 and x_2 is the same, to within a simple substitution, all the information which can be determined about the output objects will also be the same, to within a simple substitution. To prevent the possibility of interactions between abstract objects, $H\{\}$ has been restricted to be independent of all abstract objects. Although this restriction is actually stronger than is generally required, it avoids situations where an abstract signal within the specification of the system can have non-general interactions with the abstract input. For example, consider $G\{\}$ such that $G\{x[n]\} = x[n] * (w[-n]e^{-j\frac{2\pi}{N}kn})$. If the sequence $G\{w[n]\}$ is used in the determination of symmetry, then the output sequence will be found to be symmetric about zero, to within a linear-phase term. Using this information for the symmetry of $G\{x[n]\}$, when $x[n]$ is a distinct but related abstract discrete-time sequence, is incorrect.

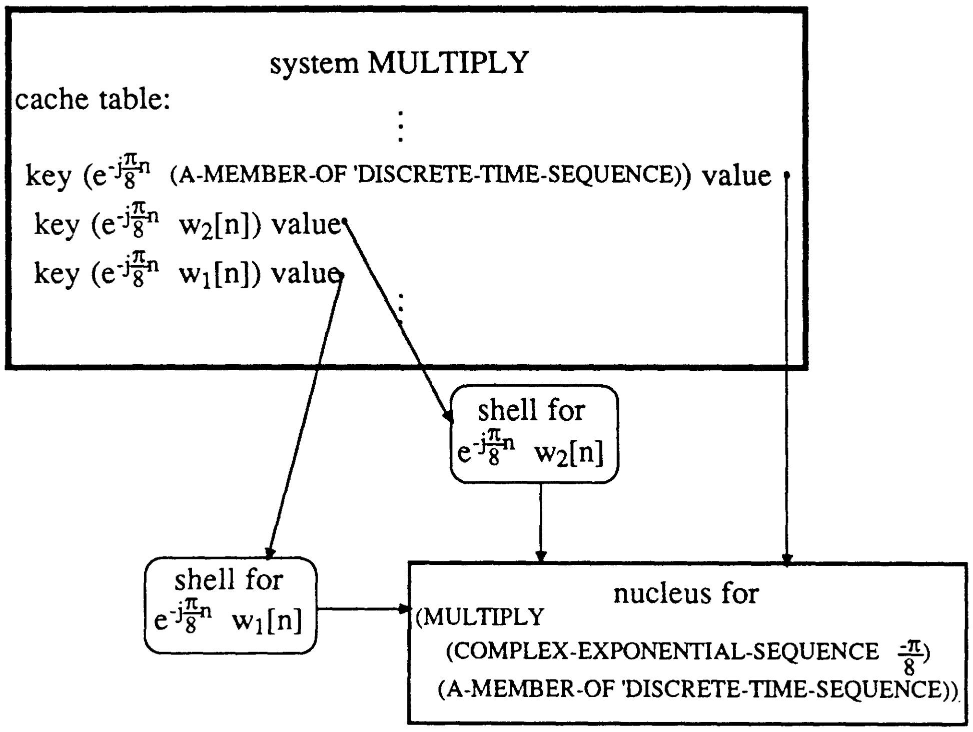
Assuming that $H\{\}$ is a simple, specific system, information determined about the output of $H\{\}$ applied to an abstract signal can be used by any output of $H\{\}$ applied to a related abstract signal. A two-level representation is again used to allow this information to be shared between related, symbolically constrained objects: Figure 4-4 shows the proposed approach pictorially.

First, consider the creation of $e^{-j\frac{\pi}{8}n}w_1[n]$, where $w_1[n]$ is an abstract, discrete-time sequence. The request for $e^{-j\frac{\pi}{8}n}w_1[n]$ prompts the MULTIPLY system for an output representation for its application to $e^{-j\frac{\pi}{8}n}$ and $w_1[n]$. Assuming that this is the first reference to a product of $e^{-j\frac{\pi}{8}n}$ with an abstract discrete-time sequence, no representation will be available within the cache table of the system MULTIPLY. Thus, a new representation must be generated. To do this, a decision must be made about how to represent the output sequence. According to the previous discussion, if there is a single dependency on an abstract object, the output sequence should be represented using a two-level representation. If there are no dependencies or multiple dependencies on abstract objects, a single-level, autonomous representation is used.[3]

To make this determination, the generating system and its inputs are examined to determine the number of dependencies on abstract objects. MULTIPLY and $e^{-j\frac{\pi}{8}n}$ are simple specific objects, independent of all abstract objects, and $w_1[n]$ is an abstract sequence which can be said to be dependent on one abstract object, namely $w_1[n]$. This

[3]Objects with dependencies on more than one abstract object are represented in the same way as simple, specific objects. To simplify the counting of the abstract objects on which these representations depend, objects with multiple dependencies on abstract objects are flagged internally in ADE.

let $w_1[n]$ and $w_2[n]$ be related but distinct instances of the abstract description (A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE)



system MULTIPLY
cache table:
key ($e^{-j\frac{\pi}{8}n}$ (A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE)) value
key ($e^{-j\frac{\pi}{8}n}$ $w_2[n]$) value
key ($e^{-j\frac{\pi}{8}n}$ $w_1[n]$) value
shell for $e^{-j\frac{\pi}{8}n}$ $w_2[n]$
shell for $e^{-j\frac{\pi}{8}n}$ $w_1[n]$
nucleus for (MULTIPLY (COMPLEX-EXPONENTIAL-SEQUENCE $\frac{-\pi}{8}$) (A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE))

Figure 4-4: Examples of symbolically constrained objects with one dependency path to an abstract instance

When a symbolically constrained object is dependent on only one abstract instance, information can be shared between the symbolically constrained objects which have the same form and are dependent on related abstract instances. This sharing of information is supported by using shells and nuclear representations. The nucleus of the representation is shared by all of the related symbolically constrained objects and the shells indicate the actual identity of the abstract instance upon which the symbolically constrained object depends. Since symbolically constrained objects are specific objects, these shells are also recorded, so that a unique representation can be maintained.

single dependence makes a two-level representation appropriate for $e^{-j\frac{\pi}{8}n}w_1[n]$. The nucleus of this representation corresponds to the application of MULTIPLY to $e^{-j\frac{\pi}{8}n}$ and the nucleus for abstract, discrete-time sequences. A shell is wrapped around this nucleus, with the shell recording the actual abstract instance, $w_1[n]$, upon which $e^{-j\frac{\pi}{8}n}w_1[n]$ depends (Figure 4-4). Both the nucleus and the shell are cached in the output signal table within the MULTIPLY system: the shell is cached under the list ($e^{-j\frac{\pi}{8}n}$ $w_1[n]$) and the nucleus is cached under the list containing $e^{-j\frac{\pi}{8}n}$ and the nucleus for abstract, discrete-time sequences (Figure 4-4). Any subsequent reference to $e^{-j\frac{\pi}{8}n}w_1[n]$ will have access to the cached shell, simply by checking in the output signal table of MULTIPLY under ($e^{-j\frac{\pi}{8}n}$ $w_1[n]$).

Thereafter, when a request is made for $e^{-j\frac{\pi}{8}n}w_2[n]$ with $w_2[n]$ being an abstract, discrete-time sequence, the previously generated nucleus will be found and reused. In more detail, in response to a request for $e^{-j\frac{\pi}{8}n}w_2[n]$, the output signal table of MULTIPLY will be examined to see if a representation is cached under the list ($e^{-j\frac{\pi}{8}n}$ $w_2[n]$). Assuming none is, the output signal table will next be checked to see if the nuclear representation is available under the list containing $e^{-j\frac{\pi}{8}n}$ and the nucleus for abstract, discrete-time sequences and the previously generated nucleus for this product will be found. A new shell will then be created and cached in the output signal table under the list ($e^{-j\frac{\pi}{8}n}$ $w_2[n]$). It is this new shell wrapped around the previously generated nucleus which will be returned as the representation of $e^{-j\frac{\pi}{8}n}w_2[n]$ (Figure 4-4).

Symbolically constrained objects with dependencies to a single abstract object, like $e^{-j\frac{\pi}{8}n}w_1[n]$ and $e^{-j\frac{\pi}{8}n}w_2[n]$, use the same approach to recording property and sample values as is used by abstract objects. The determined property values of these objects are collected, after some modification, in the nucleus of the representation. Property values which do not include the identity of the owner are recorded centrally in the nucleus without modification. Property values which do include the identity of the owner are flagged and a functional description of the property value is recorded, with a single functional argument for the identity of the abstract object upon which the owner depends (Figure 4-

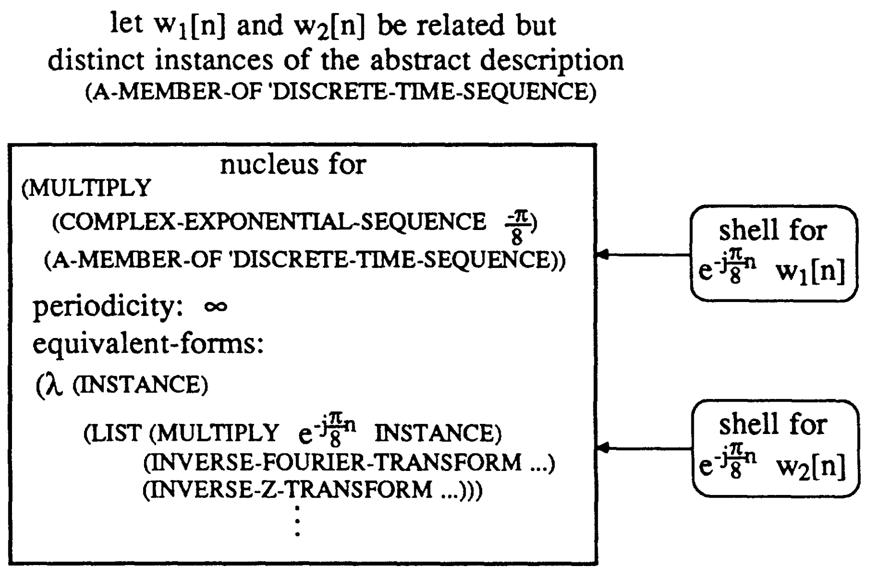

let $w_1[n]$ and $w_2[n]$ be related but
distinct instances of the abstract description
(A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE)
nucleus for
(MULTIPLY
(COMPLEX-EXPONENTIAL-SEQUENCE $\frac{-\pi}{8}$)
(A-MEMBER-OF 'DISCRETE-TIME-SEQUENCE))
periodicity: ∞
equivalent-forms:
(λ (INSTANCE)
(LIST (MULTIPLY $e^{-j\frac{\pi}{8}n}$ INSTANCE)
(INVERSE-FOURIER-TRANSFORM ...)
(INVERSE-Z-TRANSFORM ...)))
shell for $e^{-j\frac{\pi}{8}n}\ w_1[n]$
shell for $e^{-j\frac{\pi}{8}n}\ w_2[n]$

Figure 4-5: Centralization of the knowledge determined about symbolically constrained objects with one dependency path to an abstract instance
The property values of objects with a single dependency on an abstract object are collected, after some modification, in the nucleus. Property values which depend on the identity of the abstract instance are recorded in a functional form, with the identity of the abstract instance as an argument. When an attempt is made to return one of these modified property values, the result of applying the function to the particular abstract instance in use is returned instead.

5). The retrieval of a property value returns either the value recorded in the nucleus or the result of applying the functional description to the particular abstract instance. Signal sample values again must be recorded within the shell of the representation, for the same reasons as were outlined in the previous subsection.

4.3.3 Summary

This section has introduced a two-level representation for abstract objects and for symbolically constrained objects which depend on a single abstract object. Using this representation, information determined for one instance of an abstract description is reused by any instance of that abstract description. Furthermore, symbolically constrained objects with a single dependency on an abstract object share property-value information as well.

One example of information sharing between related, symbolically constrained objects was provided by lines *I-7* and *I-8* of Figure 3-4: the information determined in answering line *I-7* was reused by the related object manipulated on line *I-8*. Another example of information sharing is provided through the consecutive designs of the rectangular-window and the Hanning-window modulated filter banks, discussed in Chapter 3. In particular, the rule encoded in lines 26–38 of Figure 3-16 maps the problem of finding the equivalent forms of the two-dimensional short-time Fourier transform into the problem of finding the equivalent forms of a one-dimensional, discrete Fourier transform. Due to this mapping, the design of the rectangular-window modulated filter bank will have found all of the equivalent forms of the DFT of the product of the rectangular window with an abstract, discrete-time sequence. In the subsequent design of the Hanning-window modulated filter bank, this same rule will result in a search for the equivalent forms of the DFT of the product of the Hanning window with an abstract, discrete-time sequence. This search will also involve a search for the equivalent forms of the FFT of the product of the rectangular window with this second abstract sequence, since this is one of the subexpressions which is eventually obtained. Since the rule USING-1D-FFT generates a new abstract, discrete-time sequence on each application of the rule, the abstract sequence seen in the rectangular-window design will be distinct from the one seen in the Hanning-window design problem. It is only through the information sharing capabilities provided by the two-level representation that this search problem need not be considered a second time.

Chapter 5

Determination of Property-Value Information

The previous chapter avoided the issue of modularity in the description of property values. The desired level of modularity for property-value functions requires a finer subdivision of the signals and systems to which they are applied than can be easily provided using object-oriented programming. For example, as was seen in Figure 4-1, the Fourier transform of a product is most simply stated as a set of independent rules, each applicable to a subset of all the possible products of sequences. This gap in complexity widens when the property value is most easily described using the combination of many partial answers. For example, the set of equivalent signals is most easily enumerated by combining the answers from all the applicable rules. A single, monolithic method providing the equivalent signals would be unwieldy and extremely difficult to write.

This argument supports a finer modularity in property definitions than can be provided by methods in object-oriented programming, making rule-based definitions a natural choice. This chapter focuses on the definition and behavior of rules in a signal-processing design environment. The first section provides a description of general rule-based systems and of the specific approach used in ADE. The remainder of this chapter then focuses on some of the innovations proposed to increase the reliability and efficiency of the rule-based system used in the signal-processing environment.

5.1 Rule-based Programming in the Algorithm Design Environment

Like methods in object-oriented programming, rules in rule-based programming provide discrete descriptions which, when combined, provide a specification of some functional behavior. Rules can be defined and used in two alternate ways. Forward-chaining rules use the current state of a data base to trigger the assertion of additional information. As such, forward-chaining rules generally have one or more patterns which must be matched to trigger them and an assertion which, once triggered, they may make. An example of a forward-chaining rule is: "if the Fourier transform of a sequence exists and is known, then the inverse Fourier transform of the Fourier transform is the sequence itself." This type of behavior is particularly useful in maintaining a consistent data-base of information.

One of the characteristics of forward-chaining reasoning is its lack of focus: the computational resources are not focused on a single problem but rather are spread out in an effort to maintain the completeness and consistency of the full data base. Behavior closer to that of procedural functions is achieved using backward-chaining reasoning. Backward chaining starts when some unavailable piece of information is requested, resulting in the creation of a "goal." The backward-chaining rules are then indexed off the type of information which they can provide: the possible conclusions of the rules are used to select the rules which are appropriate for the current goal. If the conclusion matches the current goal, the preconditions of the backward-chaining rule will be examined. If the preconditions of the rule have already been met, its conclusion is asserted and the goal is achieved. Often, the validity of the preconditions of the rule will depend on other information which is unavailable and, thus, subgoals will be spawned. Examples of backward-chaining rules are shown in Figure 3-3 with the descriptions of the periodicity of a complex-exponential sequence and the simplification of the zero-frequency, complex-exponential sequence and in Figure 4-1 with the descriptions of the Fourier transforms of various products.

Two advantages of rule-based programming are its modularity and the separation of the control structure from the application-specific knowledge. Definitions are provided using the combination of numerous discrete pieces of knowledge, namely individual rules. Modularity often simplifies the description of a function: with modular definitions, independent cases can be dealt with using independent descriptions. In addition, modularity allows incremental addition to or modification of the descriptions. Finally, the modularity of the descriptions separates much of the application-specific knowledge from the control structure: the application-specific knowledge is given in the rules themselves and the control structure determines the order in which the rules are considered. In many rule-based systems, as in ADE, the behavior of the control structure can also be modified, using control strategies.

The remainder of this section considers the approach to rule-based programming taken in ADE. The majority of the signal-processing knowledge in ADE is in the form of backward-chaining rules. Therefore, these rules are considered first. Another significant part of the information within ADE is encoded using control strategies. Although ADE includes forward-chaining rules, these have not been used extensively and therefore, will be omitted from this discussion.

Backward-chaining rules in ADE commonly consist of between two and five distinct portions: Table 5.1 describes the structure of these rules. Examples of backward-chaining rule definitions are included in the signal- and system-class definitions shown in Figure 3-3 and 4-1. The definition for the COMPLEX-EXPONENTIAL-SEQUENCE class, shown in Figure 3-3, includes a description of the signal periodicities and a description of a simplification: both are translated internally into separate backward-chaining rules which are tagged as applying only to complex-exponential sequences. Similarly, the descriptions of the simplifications and Fourier transforms of sequence products shown in Figure 4-1 are translated into backward-chaining rules. Each of these rules from the definition of the

Table 5.1: Format of backward-chaining rules in ADE

Matching pattern: The rule pattern indicates the type of information which the rule can provide. If the rule pattern does not match the current goal, the rule is not considered further. If the pattern does match the current goal, the bindings from this match are enforced in all the remaining parts of the rule.

Local variables (optional): Variables for use within the rule can be defined using arbitrary functions to determine their values.

Test function (optional): An arbitrary test function can be defined. If this expression returns a null value, the rule is disqualified and not considered further. Otherwise, the rule is applicable to the current goal.

Result value: The result value given by the rule is the result of evaluating this arbitrary expression. The rule is allowed to indicate if this answer should be considered as the complete value for the current goal. Otherwise, this answer is used as a partial value and is combined with the current value of the goal.

Search termination (optional): The rule may explicitly terminate the search. This has the effect of bypassing consideration of the remaining rules and of removing any subgoals of the current goal from the schedule.

SEQUENCE-MULTIPLY-OUTPUT class is tagged as applying only to sequences belonging to that class. This type information is used along with the matching pattern and test function to determine the applicability of the rule to the goal under consideration.

Since a large part of the expertise in signal processing, as in many other fields, is knowing approaches to its problems which are fruitful, this type of information can be encoded and utilized. ADE accommodates the addition and the use of control information by providing for the definition of control strategies. The format used to represent these strategies is described in Table 5.2. Figure 5-1 shows the definitions of the two equivalent-form strategies which encode the equivalent-form behavior described in Chapter 3.

This paragraph attempts to clarify the strategy of recursively searching for the equivalent forms of an expression. To review the described behavior of this strategy, each newly uncovered equivalent form is used as a seed for another request for equivalent

Table 5.2: Format of control strategies in ADE

Matching pattern: The matching pattern indicates the type of search which the strategy is encoded to affect. If the matching pattern does not match the current goal, the strategy is not considered further. If the pattern does match the current goal, the bindings from this match are enforced in all the remaining parts of the strategy.

Local variables (optional): Variables for use within the strategy can be defined using arbitrary functions to determine their values.

Test function (optional): An arbitrary test function can be defined. If this expression returns a null value, the strategy is disqualified and not considered further. Otherwise, the strategy is taken to be applicable to the current goal.

Disallowed strategies (optional): A list of strategies which should not be considered can be provided. The listed strategies are then bypassed by the current goal and by its replacement, if a replacement goal is provided.

Replacement goal (optional): A replacement for the current goal can be provided. Providing a replacement goal has the effect of removing the current goal from consideration. Once the replacement goal is complete, the result of the replacement goal becomes the binding for the result variable of the matching pattern.

Subgoals (optional): If no replacement goal has been provided, a set of subgoals can be scheduled. Further processing of this particular strategy is suspended until all of its subgoals are completed.

Result value (optional): If no replacement goal has been provided, a result value can be given by the strategy. The strategy is allowed to indicate if this answer should be considered as the complete value for the goal. Otherwise, this answer is used as a partial value and is combined with the current value of the goal.

Search termination (optional): If no replacement goal has been provided, the strategy may explicitly terminate the search. This has the effect of bypassing consideration of the remaining strategies; of bypassing consideration of any property rules; and of removing any subgoals of the current goal from the schedule.

Access to the exhausted goal (optional): The strategy can request access to the exhausted goal. A goal is described as exhausted when all rules have been considered and all its subgoals are complete. If this access is requested, the result variable from the matching pattern is bound to the current result value of the goal. Another test function, another set of subgoals and another result value can be given for evaluation in this post-search environment.

a.

```
(DEFINE-STRATEGY RECURSIVE-EQUIVALENT-FORMS
  :GOAL (VALUE-OF ?OBJ EQUIVALENT-FORMS ?SIMPLE-EQUIVALENT-FORMS)
  :ANSWER (LIST OBJ)      ; one of the equivalent forms of an object is the object itself
  :WHEN-DONE       ; after all simple transformations are done
  :SUBGOALS      ; search for the equivalent forms of the newly uncovered forms
  ((?RECURSIVE-EQUIVALENT-FORMS
     (LET ((OLD-FORMS (GOAL-PROPERTY *GOAL* 'OLD-EQUIVALENT-FORMS)))
         ; the forms which have already been seen
       (UNLESS OLD-FORMS
         (SETQ OLD-FORMS (LIST OBJ))
         (SETF (GOAL-PROPERTY *GOAL* 'OLD-EQUIVALENT-FORMS) OLD-FORMS))
       (LET ((NEW-EQUIVALENT-FORMS      ; the newly uncovered equivalent forms
                (LOOP FOR EQUIVALENT-FORM IN SIMPLE-EQUIVALENT-FORMS
                      UNLESS (MEMBER EQUIVALENT-FORM OLD-FORMS :TEST 'SAMEP)
                      DO (PUSH EQUIVALENT-FORM (CDR OLD-FORMS))
                           ; include the new equivalent forms in the list of seen equivalent forms
                      AND COLLECT EQUIVALENT-FORM)))
         (CONS 'APPEND ; append the subgoals' answers to form a single answer
                (LOOP FOR EQUIVALENT-FORM IN NEW-EQUIVALENT-FORMS
                      COLLECT
                   (SUBGOAL '(VALUE-OF ,EQUIVALENT-FORM EQUIVALENT-FORMS
                                        ?RECURSIVE-EQUIVALENT-FORMS)
                            :ADD-PROPERTIES (LIST 'OLD-EQUIVALENT-FORMS OLD-FORMS)
                            :RECORD-ANSWER-P NIL)))))))
  :ANSWER RECURSIVE-EQUIVALENT-FORMS)
```

b.

```
(DEFINE-STRATEGY EQUIVALENT-FORMS-BY-PARTS
  :GOAL (VALUE-OF (SPECIFIC-MEMBER ?TYPE &REST ?PARTS) EQUIVALENT-FORMS
                  ?EQUIVALENT-FORMS-VALUE)
  :SUBGOALS      ; search for the equivalent forms of the components
  ((?EQUIVALENT-FORMS-BY-PARTS
     (LIST* 'OUTER-PRODUCT-OF-LISTS      ; take the outer product of the equivalent forms of the components
            (IF (TYPEP TYPE SYSTEM)
                (SUBGOAL '(VALUE-OF ,TYPE EQUIVALENT-FORMS
                                    ?EQUIVALENT-FORMS-BY-PARTS))
                '(LIST ',TYPE))
            (LOOP FOR PART IN PARTS
                  COLLECT (IF (TYPEP PART '(OR SIGNAL SYSTEM))
                              (SUBGOAL '(VALUE-OF ,PART EQUIVALENT-FORMS
                                                  ?EQUIVALENT-FORMS-BY-PARTS))
                              '(LIST ',PART))))))
  :ANSWER (LOOP FOR EQUIVALENT-TYPE-PARTS IN EQUIVALENT-FORMS-BY-PARTS
                COLLECT (APPLY 'SPECIFIC-MEMBER EQUIVALENT-TYPE-PARTS)))
```

Figure 5-1: Control strategies for recursively searching for equivalent forms and for searching for equivalent forms of subexpressions

forms. To be able to recursively start new equivalent-form searches, the control strategy RECURSIVE-EQUIVALENT-FORMS must have access to the answer after all the known simple transformations and subexpression substitutions have been completed. Thus, this access must occur after the goal has allowed all of the transformation rules to trigger and after all the subgoals of the goals are complete and have returned their answers. This additional access to the goal by the control strategy is requested through the use of the keyword :WHEN-DONE. The subgoals which are listed below this keyword in RECURSIVE-EQUIVALENT-FORMS are based on the answer which has been obtained by the equivalent-forms search to that point. Once these subgoals are completed, their answers are used to provide an additional, partial answers to the original goal. RECURSIVE-EQUIVALENT-FORMS will also trigger on these newly scheduled subgoals. Thus, the search continues recursively until no new equivalent forms are uncovered.

This section has provided a general description of the rule and control constructs which are available in ADE. The actual syntaxes which are used for encoding forward-chaining rules, backward-chaining rules and control strategies are described in Appendix A. The remainder of this chapter considers two areas of the inference structure which attempt to provide the environment with both reliability and efficiency in its determinations.

5.2 The General Characterization of Properties

For the reasons of modularity given above, explicit method-based definitions of property values are abandoned in ADE. However, method-based definitions have a number of benefits not seen with rule-based definitions. In particular, with method-based definitions, many invalid inquiries can be quickly detected: if a method is not defined for any of the classes to which the receiving object belongs, the inquiry can immediately be flagged as being invalid. In contrast, in most rule-based systems, such an inquiry would be answered using a default response. Examples of this type of error include asking for the Fourier transform of a z-transform signal or asking for the region of convergence of a discrete-time sequence.

To avoid this form of error propagation, the types of objects to which a property is applicable are enumerated and enforced. This is done by taking advantage of the object-based representation of signals and systems: an appropriately named instance variable is included in the signals and systems to which a property is applicable. The validity of queries is then easily established, simply by establishing the presence of the appropriate instance variable within the object under consideration. For example, the instance variable, FOURIER-TRANSFORM, is incorporated into the representation of discrete-time sequences and omitted from the representation of discrete-time Fourier-transform signals and z-transform signals. Thus, the validity of requests for a Fourier transform is easily checked by simply examining the object itself: the absence of the FOURIER-TRANSFORM instance variable indicates an invalid request. This addition of an instance variable also has the advantage that the instance variable can be used to record the property value, once it is determined.[1]

This approach requires an explicit declaration for each signal or system property. Table 3.2 listed the properties currently described in ADE. New property declarations are also easily added using the form DECLARE-PROPERTY, described in Appendix A. In addition to providing the list of the classes to which the property is applicable, the declaration of a property is used to shape its general behavior. These declarations include the classes of signals and systems to which the property is applicable; an initial value for the property; a function for combining the partial values with the accumulated value; and a default value to be used when no information is available about the property value. Figure 5-2 shows three examples of property declarations.

The general declaration of a property includes a list of the signal and system classes to

[1]Due to memory requirement considerations, this approach of including an explicit instance variable slot for each applicable property is only taken with the most frequently used properties in ADE. Thus, properties in ADE are classified as being "basic," if they are frequently used, or "non-basic," if they are not. "Non-basic" properties do not generate a corresponding instance variable slot. Instead, their applicability is checked using type information and their values, when determined, are cached in a general table within the owner object.

a.

```
(DECLARE-PROPERTY FT
        (DISCRETE-TIME-SEQUENCE)
  :SEED NIL
  :COMBINING-FUNCTION
    'ANSWER-AS-DONE
  :DEFAULT-VALUE
    (FOURIER-TRANSFORM-SYSTEM SELF))
```

b.

```
(DECLARE-PROPERTY ROC
        (Z-DOMAIN-SIGNAL)
  :SEED NULL-INTERVAL
  :COMBINING-FUNCTION
    'INTERVAL-COVER
  :DEFAULT-VALUE UNKNOWN)
```

c.

```
(DECLARE-PROPERTY NON-ZERO-SUPPORT
        (DISCRETE-TIME-SEQUENCE FOURIER-DOMAIN-SIGNAL Z-DOMAIN-SIGNAL 2D-SEQUENCE)
  :SEED (INTERVAL −∞ ∞ (CONTINUOUS-DOMAIN-P SELF))
  :COMBINING-FUNCTION 'INTERVAL-INTERSECT
  :DEFAULT-VALUE (INTERVAL −∞ ∞ (CONTINUOUS-DOMAIN-P SELF)))
```

Figure 5-2: Examples of the declaration of properties

which the property is applicable. This allows the environment to include an appropriate instance variable in the signals and systems which fall within these classes.

Since the values of most properties are formed by combining information from multiple sources, a function for combining the partial answers is provided. To illustrate the necessity for an explicit description of the combination function, consider the region of convergence of a z-transform signal and the non-zero support of a discrete-time sequence. The region of convergence of a z-transform signal includes all the regions of the z-plane on which the z transform converges. Thus, multiple assertions about regions of convergence are combined using the smallest interval covering all the asserted regions (Figure 5-2-b).[2] For the non-zero support of a discrete-time sequence, any asserted support must contain the full non-zero support. Thus, multiple assertions about the support are combined using the intersection of all the asserted non-zero supports (Figure 5-2-c). These two examples illustrate the diversity of combination functions, even for two properties whose

[2]In ADE, the region of convergence is indicated either using a continuous interval, to represent the radial extent of the ROC or using #<UNKNOWN>, to represent an unknown ROC.

values are both intervals. The provision for an initial value allows all partial answers to be treated identically, whether or not the partial answer is the first piece of information to be received.

Functions for combining partial answers can also be used to terminate the search for a property value. By having the combination function flag answers which can not be further modified, the remaining available information is not considered. Some examples of property values which can not be further modified include an empty interval for a non-zero support; a region of convergence between 0 and ∞; and a single point for the range of a signal. For property values where any "partial answer" is actually the value for the property, the combining function can flag every answer as being complete. An example of this is the Fourier transform of a signal (Figure 5-2-a): any expression for the transform is a complete description.

An explicit default value is provided, to be used when no information is available about the property value. Often the default value is the same as the initial value but occasionally their values are distinct. An example of distinct initial and default values is the region of convergence of a z-transform signal. In determining the ROC of a z-transform signal, initially, no region is known to converge. Since the ROC combining function uses the covering interval, the initial value must be the empty interval. If the search terminates without finding any information about the ROC, then the unique value #<UNKNOWN> is returned to indicate this lack of knowledge (Figure 5-2-b). The empty interval can not be used since there is a distinction between stating that a z-transform signal does not converge and stating that the region of convergence is unknown. This distinction is highlighted by considering the z transform of an abstract discrete-time sequence. Obviously, in this case, no information can be given about the ROC of the z-transform signal. Nor can it be said that the z transform does not converge, since this statement also imposes certain constraints on the discrete-time sequence, constraints which can not be imposed using the given information. So, #<UNKNOWN> is returned to indicate this lack of information.

This section has discussed the explicit declaration of properties as a way to retain the advantages of method-based definitions while still having the modularity of rule-based definitions. The remainder of this chapter focuses on increasing the efficiency of the determination of property values.

5.3 Efficiency in Property-Value Determination

Another important issue in property definitions is the efficiency with which the property value can be determined. Unfortunately, the modularity of rule-based systems and their separation of the control structure from the the application-specific knowledge often result in reduced computational efficiency. Some of this inefficiency can be avoided by allowing the combining function for partial answers to terminate a search, as described above. The remainder of this chapter focuses on other ways of increasing the efficiency of property-value determinations.

5.3.1 Static hierarchical organization of the rule base

To increase the efficiency of the rule-based system in a signal-processing design environment, patterns in common queries can be exploited. In particular, the most commonly derived type of information in signal processing is the value of a particular property of a known signal or system. This trend is exploited in ADE to increase the efficiency of the search for a property value by altering the static layout of the rule base.

The rule base in ADE is organized hierarchically. Rules for determining property values are collected within the most restrictive signal or system class which contains all the objects to which the rule may apply. For example, the rule shown on lines 14–33 of Figure 3-13, for pulling common shifts to the outside of a generalized shift-invariant system, is associated with the signal class for the outputs of generalized shift-invariant systems. In this way, the search to determine a property value need only consider those rules associated with the classes and types to which the current object belongs.

The advantages of this approach are twofold: the number of rules considered is reduced and the number of explicit tests in the rules themselves may be reduced. The number of rules considered is reduced since only the rules associated with the classes of the current object are considered. The number of tests required in a given rule may also be reduced. Since rules will only be tested against objects belonging to the class or type with which the rule is associated, explicit tests for this membership can be omitted. Again, using the example of the generalized shift-invariant system output, no test is required to assert that the signal under consideration is the output of a generalized shift-invariant system nor is a test required to assert that its generating system is a generalized shift-invariant system. Removing some of the tests used in determining the applicability of a rule obviously reduces the cost of testing the rule for applicability.

Continuing in the same vein, property-value rules are also sorted within a single class according to the property. Thus, the rules for the non-zero support will not be indexed in a search for the equivalent forms of a signal. Adding this second dimension to the rule hierarchy also reduces the number of rules considered.

5.3.2 Efficient descriptions for backward-chaining rules in signal processing

Another area which is used in ADE to increase the efficiency of property-value determination lies in the internal layout of the rules themselves. Typically, the applicability of a signal-processing rule will depend on the type of the object and on the components making up that object. The efficiency of matching and firing a rule can be easily affected by the location of these typing and component restrictions: assuming these restrictions are included in the matching pattern, these constraints can be used to quickly reveal mismatches between the rule and the goal.

By providing typing information directly in the matching variables, any mistyping can be determined when the binding between the matching variable and an object is first created and further consideration of that rule can be avoided. For example, consider the simplification rule shown in Figure 3-13. The rule pattern includes a test to insure that the generating system is not itself a shift:[3] otherwise, two cascaded shift systems could be interchanged forever. Obviously, this type restriction could be included in the test function of the rule. By instead associating the type test with the matching variable itself, the test can be made when the match is first being attempted. In this way, mismatches between the rule and the current goal can be detected before the full match is attempted and before any of the values for the local variables are determined.

Similarly, subpattern matching can be used to quickly reveal mismatches between the rule and a goal. Consider the rule for simplifying the product of inverse z transforms (lines 11–19 of Figure 4-1). This rule will be tested in any attempt to simplify the FSK-code frequency chip $w[n]e^{-j\frac{2\pi}{16}n}$. By immediately attempting to match against the components, a mismatch between the subpattern and the multiplicands can be used to quickly abort consideration of that rule.

Subpattern matching raises another issue which can strongly affect the efficiency of determining the the applicability of a rule: namely, matching against the inputs to operators which are commutative and associative. To illustrate, consider the process of determining the Fourier transform of the modulated Hanning-window FSK-code frequency chip $r_{16}[n](1 - \cos(\frac{2\pi}{16}n))e^{-j\frac{2\pi}{16}n}$. The search for applicable rules will eventually attempt to match the frequency chip against the object pattern for the rule shown on line 28–40 of Figure 4-1. Given the topology of the two, the first attempt to match fails, as shown in Figure 5-3, since $r_{16}[n]$ does not match the pattern for the complex-exponential sequence. In order to achieve a match, the multiplicands in the FSK-code frequency chip must be permuted. As shown in Figure 5-3, the third permutation results in a match. Thus, when systems are commutative and associative, all permutations of inputs must be eliminated before a mismatch between the object and the pattern can be declared. The number of permutations which potentially must be considered is the factorial of the number of inputs. For example, in considering the summations within the incoherent combination

[3]Shift-invariant systems of one input, such as a shift system, are also defined as generalized shift-invariant systems in ADE.

```
(FOURIER-TRANSFORM
  (SEQUENCE-MULTIPLY
    (CAUSAL-RECTANGULAR-WINDOW 16)
    (SEQUENCE-SUBTRACT 1 (COSINE-SEQUENCE ($/ 2PI 16)))
    (COMPLEX-EXPONENTIAL-SEQUENCE ($/ 2PI 16))))
                    ⋮
; determine applicability of the sequence-multiply-output Fourier-transform rule
; COMPLEX-EXPONENTIAL-MODULATED-INPUT
;
;  attempt to match the pattern
;  (OUTPUT-OF ?MULTIPLY (SPECIFIC-MEMBER COMPLEX-EXPONENTIAL-SEQUENCE ?FREQ) &REST ?OTHER-INPUTS)
;  against the object $r_{16}[n](1-\cos(\frac{2\pi}{16}n))e^{-j\frac{2\pi}{16}n}$
;
;    bind ?MULTIPLY to #<SEQUENCE-MULTIPLY>
;
;      mismatch between (SPECIFIC-MEMBER COMPLEX-EXPONENTIAL-SEQUENCE ?FREQ) and $r_{16}[n]$
;    permute the multiplicands and attempt match against reordered multiplicands
;      mismatch between (SPECIFIC-MEMBER COMPLEX-EXPONENTIAL-SEQUENCE ?FREQ) and $(1-\cos(\frac{2\pi}{16}n))$
;    permute the multiplicands and attempt match against reordered multiplicands
;      bind ?FREQ to 0.39269908169872414d0
;
;  match between pattern and $r_{16}[n](1-\cos(\frac{2\pi}{16}n))e^{-j\frac{2\pi}{16}n}$
;
; the sequence-multiply-output Fourier-transform rule COMPLEX-EXPONENTIAL-MODULATED-INPUT
; is applicable to $r_{16}[n](1-\cos(\frac{2\pi}{16}n))e^{-j\frac{2\pi}{16}n}$
                    ⋮
```

Figure 5-3: Pattern matching against the inputs of a commutative, associative operator

Subpattern matching is used to quickly reveal mismatches between the rule and a goal. However, when the object under consideration is the output of a commutative, associative operator, the components matched against its inputs must consider the permutations of those inputs before declaring a mismatch. This necessity is illustrated here. It is only on the third permutation of the FSK-code frequency chip $r_{16}[n](1-\cos(\frac{2\pi}{16}n))e^{-j\frac{2\pi}{16}n}$ that a match is achieved between the object and the pattern for the Fourier-transform rule shown on lines 28–40 of Figure 4-1.

of the 16-channel FSK-code detector, the number of permutations of the addends which may need to be considered is over 2×10^{13}. The possibility of matching against each of these permutations must be eliminated before a mismatch may be declared.

The intuitive solution to this problem is to exploit the information provided by the discovered mismatches, using a mismatch against one permutation to remove some of the remaining permutations from consideration. For example, if a mismatch between the pattern and the object is detected when attempting to match one of the inputs against the first part of the subpattern, then all the other permutations with that input first can be removed from consideration. Thus, since a mismatch between $r_{16}[n](1 - \cos(\frac{2\pi}{16}n))e^{-j\frac{2\pi}{16}n}$ and the pattern shown in Figure 5-3 is detected on consideration of $r_{16}[n]$ and the first part of the subpattern, any permutation with $r_{16}[n]$ first is known to result in a mismatch, so that the permutation $r_{16}[n]e^{-j\frac{2\pi}{16}n}(1 - \cos(\frac{2\pi}{16}n))$ need not be considered.

Furthermore, when some or all of the parts of the subpattern impose identical constraints, then permutations within the inputs matching that part of the pattern need not be considered. This uniformity in constraints often occurs through the use of &REST: this keyword is used to match a single variable against the remaining subexpressions of the object. Examples of this are provided on lines 7–13 and lines 14–33 of Figure 3-13 and on lines 11–19 and lines 28–40 of Figure 4-1.[4] Permuting the set of inputs which are matched by this single variable without changing the members of the set has no effect on the success or failure of the matching process. Thus, by exploiting this partial insensitivity to order, the number of permutations which must be considered can be reduced.

The preceding discussion has attempted to highlight ways of increasing the efficiency of property-value determinations. The savings that are possible simply by modifying the matching process are significant and, in the case of rules for the outputs from commutative

[4]Most of these examples include an "element subpattern" which is used to impose additional constraints on the elements of the lists to which the &REST variable is matched. "\${" is used to start an element subpattern and "}" then closes that subpattern. Matching variables within the subpattern must be bound to a single instance across all the elements, unless the notation "?[*variable*]" is used to allow diversity. Thus, ?INPUTS\${(OUTPUT-OF ?SHIFT ?[SHIFT-INPUTS])} will match a list of outputs from a single shift system applied to any combination of inputs.

and associative systems, the increase in efficiency is imperative if the consideration of many signal-processing problems are to be in the realm of possibility.

5.4 Summary

This chapter has outlined a rule-based environment, tailored to the needs of symbolic manipulation of signal-processing expressions. The rule-based paradigm is required to provide the desired level of modularity to the property-value definitions. This chapter has focused on increasing the reliability and efficiency of property-value determinations. Reliability was improved by requiring explicit declarations of signal and system properties. These declarations are used both to enumerate the classes of the objects to which a property is applicable and to describe the general characteristics of the property, such as the function for combining partial answers and a default value. Efficiency was improved by organizing the rule base hierarchically and by modifying the matching process. By organizing the rule base hierarchically, according to the class of objects to which each rule applies, the number of rules which will be considered for any given goal is reduced. The class structure provided by object-oriented definitions provide a natural hierarchy for this organization.

The remainder of the discussion in this chapter has centered on increasing the efficiency of rule testing. These efforts concentrated on the testing stage, since a large percentage of the computational resources will be devoted to the testing of rules: all of the indexed rules must be tested on a goal, but generally only a small fraction of them are actually fired. Matching variables include both type constraints and subpattern constraints. Two approaches were described for improving the efficiency of matches against the inputs to commutative, associative operators. In particular, the information provided by a mismatch using one permutation is used to remove other permutations from consideration and permutations are not considered within the elements which match a single, uniform set of constraints.

Chapter 6

Regularity in Signal-Processing Algorithms

One of the primary tasks of an algorithm design environment is to explore the space of implementations of a given input/output mapping. In order to explore the full design space, the environment must search through the space of algorithms with the same transfer characteristics. Chapter 3 described the process by which these transformations are found. In particular, all of the transformations which are directly applicable to the original signal-processing expression or to any of its subexpressions are completed to generate alternate implementations. Any newly uncovered implementations and their subexpressions are then themselves used as seeds to find further transformations. This process repeats until no new implementations are found. Figure 5-1 showed the control strategies which encode this extended search.

Two major difficulties with this search process become apparent after careful consideration: the possibility of the infinite expansion of the search space and the finite but combinatorial growth of the space due to the separate manipulation of subexpressions. The search space can expand infinitely by two different paths. A simple transformation or a combination of simple transformations can result in the introduction of identity operations. Alternately, the decomposition of expressions into subexpressions can create a processing loop. Both of these difficulties are considered briefly in this chapter. The problem of limiting the combinatorial growth is then the subject of the remaining sections of this chapter.

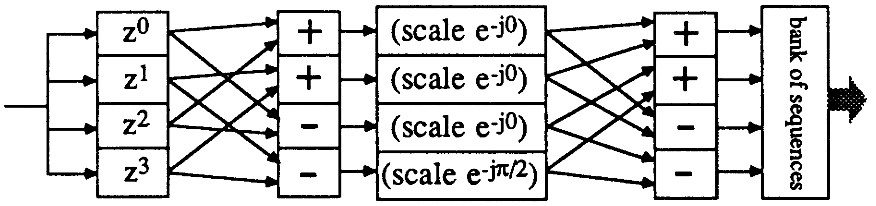
6.1 Growth of the Design Space for Signal-Processing Algorithms

6.1.1 Infinite expansion of the design space due to increasing complexity introduced by simple transformations

The transformations used in generating equivalent implementations often result in signal-processing algorithms whose complexity is greater than the manipulated algorithm. For example, consider the problem of finding equivalent implementations of the matched filters for the frequency chips in the rectangularly windowed, FSK-code detector. One of the rearrangements which is found is shown in Figure 6-1-a. From the description of the search process given in Chapter 3, the subexpressions of this implementation will be manipulated, as will the subexpressions of each new implementation. Using this approach without modification will result in the formation of increasingly complex expressions and, ultimately, an infinite expansion of the search space. The source of the increasing complexity is the rule shown on lines 14–33 of Figure 3-19. This rule attempts to pull common scaling factors through a generalized homogeneous system. The rule is applicable to some of the subexpressions of the structure shown in Figure 6-1-a: Figure 6-1-b shows the result of applying this rule to each of these subexpressions. In fact, the rule is also applicable to the subexpressions of Figure 6-1-b. As a result of the recursive search, these applications would also be completed. These recursive applications and the fact that each of these applications increases the complexity of the algorithmic description would then result in the infinite expansion of the search space.

As can be seen from this example, care must be taken to limit the complexity of

a.



b.

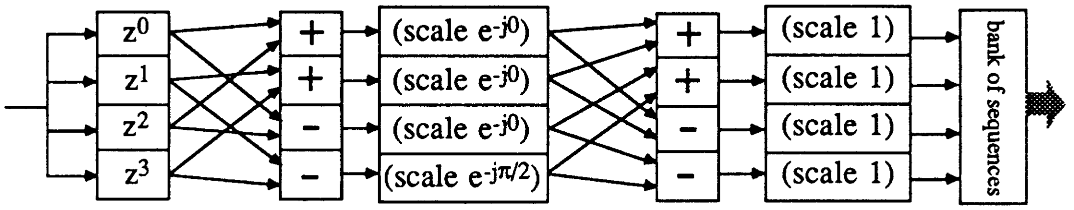


Figure 6-1: An example of increasing complexity resulting from equivalent-form manipulations

Transformations used in generating equivalent implementations often result in signal-processing algorithms whose complexity is greater than the manipulated algorithm. As an example, one of the transformations completed in finding the equivalent forms of the matched filters for the 4-point rectangular-window frequency chips in the FSK-code detector is shown here. The rule for pulling common scaling operations through a generalized homogeneous system, shown in Figure 3-19, is applicable to some of the subexpressions of the structure shown in part (a): part (b) shows the result of applying the rule to each of these subexpressions. In fact, this rule is also applicable to some of the subexpressions of the expression in part (b).

the manipulated algorithms before they are used as seeds to find additional equivalent forms. Simplifications are used to control the complexity of the signal-processing expression. SIMPLIFICATION, when applied to a signal-processing expression, returns the simplest direct description of the expression that the environment can find. The simplest description of a signal-processing expression is obtained both by simplifying its subexpressions, using the strategy shown in Figure 6-2-a, and by recursively simplifying the modified description, using the strategy shown in Figure 6-2-b. The actual simplifications are encoded using over 300 of the 850 rules currently included in ADE: examples

a.

```
(DEFINE-STRATEGY SIMPLIFICATION-BY-PARTS
  :GOAL (VALUE-OF (SPECIFIC-MEMBER ?TYPE &REST ?PARTS) SIMPLIFICATION
                  ?SIMPLIFICATION-VALUE)
  :REMOVE-STRATEGIES      ; only run this once on each goal/replacement goal pair
  (SIMPLIFICATION-BY-PARTS)     ; removes strategy from consideration on the replacement goal
  :REPLACEMENT-GOAL      ; simplify the expression using simplified components
  (SUBGOAL '(VALUE-OF ,(APPLY 'SPECIFIC-MEMBER (SIMPLIFICATION TYPE)
                              (MAPCAR 'SIMPLIFICATION PARTS))
                      SIMPLIFICATION ?SIMPLIFICATION-VALUE)))
```

b.

```
(DEFINE-STRATEGY RECURSIVE-SIMPLIFICATION
  :GOAL (VALUE-OF ?OBJ SIMPLIFICATION ?SINGLE-SIMPLIFICATION)
  :WHEN-DONE      ; after all "simple" simplifications are done
  :WHEN (AND SINGLE-SIMPLIFICATION (NOT (EQ OBJ SINGLE-SIMPLIFICATION)))
      ; if a simplification has been found
  :SUBGOALS      ; try to find any further simplifications
  ((RECURSIVE-SIMPLIFICATION
    (SUBGOAL '(VALUE-OF ?SINGLE-SIMPLIFICATION SIMPLIFICATION
                        ?RECURSIVE-SIMPLIFICATION))))
  :SET-ANSWER RECURSIVE-SIMPLIFICATION)
```

Figure 6-2: The control strategies for finding simplifications
The first strategy attempts to simplify an expression by first simplifying its subexpressions. The second strategy provides recursive simplification of an object.

of SIMPLIFICATION rules can be found in Figures 3-3 and 4-1.

As stated above, the complexity of the manipulated algorithms must be limited, through simplification, before the algorithms are used as seeds in the recursive search for equivalent implementations. This intermediate step of simplification is neatly and easily included in the search process by including it in the combining function provided for EQUIVALENT-FORMS, as shown in Figure 6-3. The combination function used by EQUIVALENT-FORMS simplifies the equivalent implementations before including them in the equivalent-forms answer. This simplification occurs before the strategy RECURSIVE-EQUIVALENT-FORMS sorts the implementations to find new seeds for further searches and,

```
(DECLARE-PROPERTY EQUIVALENT-FORM
          (SIGNAL SYSTEM)      ; applicable to all signals & systems
  :SEED (LIST (SIMPLIFICATION SELF))      ; one equivalent form is the simplified object itself
  :COMBINING-FUNCTION 'SIMPLIFY-AND-INSERT      ; see below
  :DEFAULT-VALUE (LIST (SIMPLIFICATION SELF)))

(DEFUN SIMPLIFY-AND-INSERT (NEW-FORMS PREVIOUS-FORMS)
      ; NEW-FORMS are the forms which are being added & PREVIOUS-FORMS are the previously uncovered forms
  (LOOP FOR FORM IN NEW-FORMS
          AS SIMPLIFIED-FORM = (SIMPLIFICATION FORM)
              ; simplify each of the new forms before considering it further
          UNLESS (MEMBER SIMPLIFIED-FORM PREVIOUS-FORMS :TEST 'SAMEP)
              ; if this form has not been seen before, add it
            DO (PUSH SIMPLIFIED-FORM PREVIOUS-FORMS)
          FINALLY (RETURN PREVIOUS-FORMS)))
```

Figure 6-3: The property declaration for EQUIVALENT-FORM

thus, the explosive expansion of the search space due to arbitrarily complex algorithmic descriptions is avoided.

6.1.2 Infinite expansion of the design space due to identity loops in recursive subexpression decomposition

As described earlier, the search for the equivalent implementations of a signal-processing expression explicitly searches for the equivalent implementations of the subexpressions as well. Thus, as was illustrated in Figure 3-5 and Figure 5-1, subgoals are generated, requesting the equivalent implementations of the inputs to the outer processing layer of the expression. This process recursively decomposes the signal-processing expression until the remaining inputs are can not be further decomposed. An infinite recursion will occur if one of these subgoals, coincides with the original search space. Figure 6-4 provides an example of this difficulty.

This coincidence between the search space for the original signal-processing expression

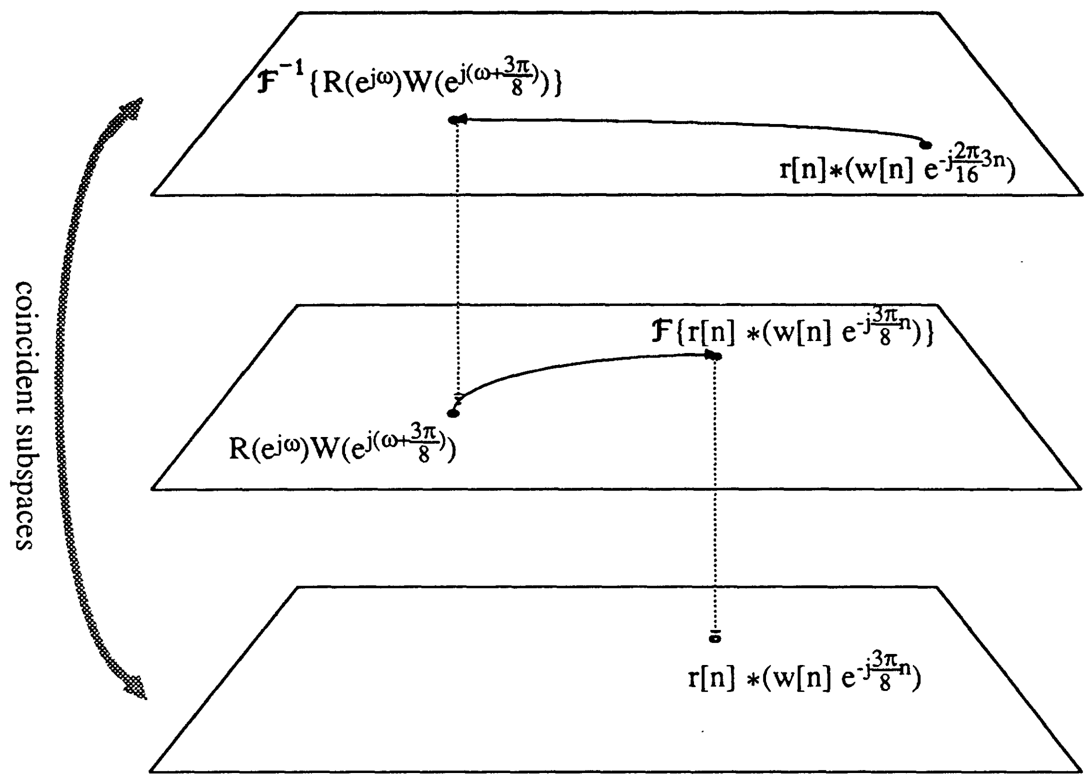

$\mathcal{F}^{-1}\{R(e^{j\omega})W(e^{j(\omega+\frac{3\pi}{8})})\}$
$r[n]*(w[n]\,e^{-j\frac{2\pi}{16}3n})$
$\mathcal{F}\{r[n]*(w[n]\,e^{-j\frac{3\pi}{8}n})\}$
$R(e^{j\omega})W(e^{j(\omega+\frac{3\pi}{8})})$
$r[n]*(w[n]\,e^{-j\frac{3\pi}{8}n})$
coincident subspaces

Figure 6-4: The occurrence of identity loops in the search for equivalent forms due to the coincidence of a subexpression search with a superior search space

In searching for the equivalent forms of a signal, the equivalent forms of the inputs to the generating function are used to give additional equivalent forms for the signal itself. Thus, the subordinate goals will be generated, requesting the equivalent forms of the inputs signals. Occasionally, one of these subordinate goals will coincide with the original goal: this occurs when the composite operation to go from the coinciding subgoal space back to the original goal space is an identity transform. An example of this is shown here.

```
(DEFINE-STRATEGY BREAK-EQUIVALENT-FORMS-IDENTITY-LOOPS
  :GOAL (VALUE-OF ?OBJ EQUIVALENT-FORMS ?EQUIVALENT-FORMS-VALUE)
  :WITH       ; look up the goal tree to see if this same goal is a supergoal
  ((SUPERIOR-GOAL-WITH-SAME-FORM
     (SUPERIOR-GOAL-FORM-P *GOAL* (GOAL-FORM *GOAL*))))
  :WHEN SUPERIOR-GOAL-WITH-SAME-FORM      ; if this same goal is a supergoal
  :REMOVE-STRATEGIES       ; disallow these strategies
  (RECURSIVE-EQUIVALENT-FORMS EQUIVALENT-FORMS-BY-PARTS)
  :DO      ; prevent the goals in the loop from recording their answers
  (LOOP FOR SUPERIOR IN (PATH-BETWEEN-TWO-GOALS
                          *GOAL* SUPERIOR-GOAL-WITH-SAME-FORM)
        DO (SETF (GOAL-RECORD-ANSWER-P SUPERIOR) NIL)
        FINALLY (SETF (GOAL-RECORD-ANSWER-P *GOAL*) NIL))
  :SET-ANSWER (LIST OBJ)      ; set the answer to simply the object under consideration
  :DONE)      ; terminate consideration of this goal
```

Figure 6-5: The control strategy for breaking identity loops

In searching for equivalent forms of a signal, identity loops are occasionally created by coincidence between the search spaces of a subexpression and a superior expression. The strategy shown here detects and breaks these identity loops, thus avoiding infinite recursions.

and the search space for the subexpression creates a loop. Unless these loops are broken before the decomposition process continues, the recursive decomposition will result in an infinite unwinding of the loop. This infinite expansion of the search can not be avoided by simplification, since each of the subexpressions which are being considered are themselves completely simplified. Therefore, an explicit control strategy, shown in Figure 6-5, is provided to prevent this infinite recursion. This strategy prevents the goals that lie within the loop from recording their answers, since the answers to these goals are affected by the break in the loop, and it terminates the current goal. This breaks the identity loop. Once the equivalent forms of the subexpressions are projected back into a search space outside the loop, the answers are unaffected by the break in the loop and, therefore, they are recorded.

6.1.3 Combinatorial growth of the algorithm design space

The search for equivalent implementations of a signal-processing expression must consider the equivalent implementations of the subexpressions as well as the complete expression itself. Since each of the subexpressions are manipulated independently and independently recombined to form new equivalent expressions, the size of the search space under consideration grows combinatorially with the number of subexpressions.

One possible strategy for limiting this combinatorial growth in searches for efficient implementations relies on the cost measure of each subexpression to heuristically prune the space. Instead of enumerating all the equivalent implementations of a signal-processing expression and then filtering out the inefficient and uncomputable structures using the overall cost measure, this strategy would immediately prune the number of subexpression implementations, prior to their upward propagation, based on their relative costs. This approach relies on the assumption that, when propagating two subexpression implementations upward, the more expensive subexpression implementation will not be incorporated into any of the efficient implementations. Unfortunately, this pruning strategy suffers from the interaction of subexpression costs: the cost of using one form of an input is often ameliorated by the use of the same form for other inputs. Thus, the contribution of a subexpression to the overall cost of an enclosing expression is not independent of the other parts of the enclosing expression. Furthermore, although each individual transformation tends to be local, the recursive approach to finding equivalent forms can result in extensive, global transformations: pruning out a particular form before it is considered as an input to subsequent levels of processing may prevent the derivation of another globally-altered, lower-cost implementation.

The approach to limiting the search space which is advocated in this thesis exploits the internal regularity of signal-processing algorithms. By enforcing internal correspondences, the space of equivalent forms which will be explored can be drastically reduced. Furthermore, the structure of the resulting algorithms should be more appropriate for parallel processing than the unstructured algorithms which result from a full search.

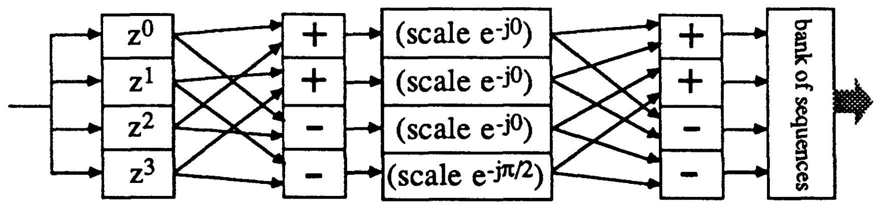


Figure 6-6: The short-time Fourier transform structure as an example of a signal-processing algorithm with a regular internal structure

Many signal-processing algorithms have a highly regular internal structure. One example of internal regularity was given in Figure 3-6, with the incoherent combination function. Another is shown here, with the 4-point short-time Fourier transform. Other structures with internal regularity include the polyphase implementation of a downsampled convolution and the polyphase implementation of a convolution with an upsampled input.

6.2 Regularity in Computation

Many signal-processing algorithms have a highly regular internal structure. Most parallel implementations of these algorithms rely on this internal regularity. For example, the internal regularity of local, shift-invariant, image-processing operations, such as morphological operations, can be easily exploited in mappings onto SIMD architectures: the regularity of these operations, embodied in the self-similarity of the operations on distinct samples, allows many of the operations to be described using a single instruction. Many other signal-processing algorithms exhibit a more general type of internal regularity, which will be examined in this section.

In Chapter 3, Figure 3-6 was used to point out the regularity in the incoherent combination function of the FSK-code detector. Figure 6-6 provides another example of a signal-processing algorithm with a highly regular internal structure, namely the 4-point short-time Fourier transform using a rectangular window. The signal flow graph shown in Figure 6-6 itself implicitly uses the regularity of the algorithm: signals which are used multiple times are only shown once, with their dependencies fanning out from a single point. Visually, this suggests that the implementations of these multiple uses of the signals will always be coincident. On the contrary, unless a correspondence constraint imposes correspondence on the multiple uses, the implementations for the separate uses will be selected separately. Conceptually, this breaks the implied coupling of the butterflies: these operations would no longer form butterflies since they would not use the same inputs. Instead, the short-time Fourier transform would be computed using a tree structure, like that shown in Figure 6-7.

In order to maintain the regularity of the short-time Fourier transform structure, three sets of constraints are imposed on the low-level description shown in Figure 6-7. The similarity of the sequences feeding into the BANK-OF-SEQUENCES is pointed out using a "correspondence constraint": the manipulation of these expressions and their subexpressions are constrained to occur in synchrony. In addition, the sequences feeding into each of the outer SEQUENCE-ADD/SEQUENCE-SUBTRACT systems are similar: that is, the first addend into the k'th system is similar to the second addend into the k'th system. Thus, a second set of correspondence constraints is placed on the inputs into each of the outer SEQUENCE-ADD/SEQUENCE-SUBTRACT systems. The final level of regularity in this structure occurs at each of the inner SEQUENCE-ADD/SEQUENCE-SUBTRACT systems: a third set of correspondence constraints is placed on the inputs into each of these SEQUENCE-ADD/SEQUENCE-SUBTRACT systems. Through these correspondence constraints, the manipulation of the corresponding subexpressions of the short-time Fourier transform is constrained to occur in synchrony, resulting conceptually in the manipulation of

$$Y[n_1, n_2] = S_{n_2}[n_1]$$
$$S_k[n] = \sum_{i=0}^{1} \pm R_{(k \bmod 2),i}[n] \text{ (using "+" on } k = 0,1 \text{ and "−" on } k = 2,3)$$
$$R_{l,i}[n] = e^{-j\frac{2\pi}{4}il} \times P_{l,i}[n]$$
$$P_{l,i}[n] = \sum_{m=0}^{1} \pm x[n + 2m + i] \text{ (using "+" on } l = 0 \text{ and "−" on } l = 1)$$

Using constrained manipulations, the uncovered implementations will also have a regular internal structure. Without the correspondence constraints, the regular internal

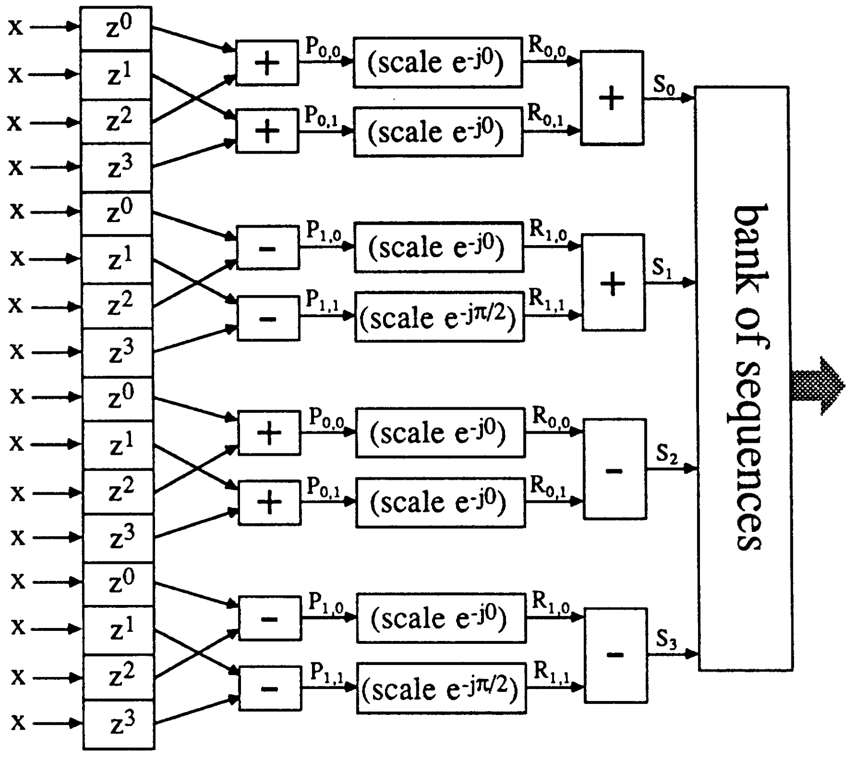


Figure 6-7: The short-time Fourier transform structure without any implied couplings between subexpressions

The signal flow graph shown in Figure 6-6 implicitly uses the coincident structure of the FFT algorithm: signals which are used multiple times are only shown once, with their dependencies fanning out from a signal point. Visually, this suggests that the implementations of these multiple uses of the signals will always be coincident. On the contrary, unless a correspondence constraint imposes correspondence on the multiple uses, the implementations for the separate uses are selected separately. The signal flow graph for the transform structure is shown here without these visual couplings. Without correspondence constraints, the butterflies of the FFT will not generally be implemented as butterflies, since the implementations of their inputs are uncoupled.

structure would be lost and the size of the design space would explode.

6.3 Expressing and Maintaining Regularity

In the previous section, the short-time Fourier transform structure was examined to illustrate the type of internal regularity seen in many signal-processing algorithms. Correspondence constraints are imposed by the operator MAP-OVER to maintain this regularity: an example of the use of MAP-OVER has already been provided by the definition of the INCOHERENT-COMBINATION, shown in Figure 3-2. The signal generated by MAP-OVER is a composition operator: for example, the 4-channel incoherent combination of a two-dimensional sequence FILTER-OUTPUTS relies on the signal shown in Figure 3-6 to provide a description of its sample values and of many of its properties.

Once a correspondence constraint is imposed on a structure, ADE propagates the correspondence inwards, through the inputs to systems and the parameters of classes. Furthermore, ADE propagates the correspondence constraints to the Fourier and z transforms of the constrained structure and to any of the equivalent expressions which are generated in searches for CONSTRAINED-SIMPLIFICATION and CONSTRAINED-EQUIVALENT-FORMS.

Searches for most property values, such as NON-ZERO-SUPPORT or SAMPLE-TYPE, are unaffected by correspondence constraints: their manipulations of parallel expressions continue independently. The correspondence constraints only affect searches for transforms and for equivalent expressions. Furthermore, searches for equivalent expressions can be made to respect correspondence constraints or to ignore these constraints. CONSTRAINED-SIMPLIFICATION and CONSTRAINED-EQUIVALENT-FORMS, which respect correspondence constraints, use the same rules as SIMPLIFICATION and EQUIVALENT-FORMS, which ignore them: it is simply the combination of these manipulations on subexpressions which is distinct. This distinction will be discussed in detail later in this section.

This section discusses the inward propagation of correspondence constraints; the propagation of correspondence constraints to modified implementations; and the interactions between the correspondence constraints and constrained searches.

6.3.1 Propagating correspondence constraints through parallel expressions

Once a correspondence constraint is imposed, using MAP-OVER, that constraint propagates inwards until a mismatch is detected, affecting the transform manipulations and constrained equivalent-expression manipulations of all the intervening expressions. This subsection examines the manipulations which must be done on parallel expressions to find constrained equivalent expressions. A similar process occurs in searches for transforms and for constrained simplifications on parallel expressions.

The approach used to find unconstrained equivalent forms was described in Chapter 3. This approach was encoded in the two strategies shown in Figure 5-1. The first of these strategies (Figure 5-1-a) used newly uncovered equivalent forms as seeds for recursive requests for additional equivalent forms. This same strategy is used, without modification, in searches for constrained equivalent forms. The second strategy used for obtaining equivalent forms (Figure 5-1-b) finds equivalent forms of a signal by finding the equivalent forms of each of the inputs separately and then using these forms to replace the original set of inputs. A similar but modified strategy exists for finding constrained equivalent forms. In particular, some of the constrained equivalent forms of a signal or system are found by replacing the generating components by their constrained equivalent forms. If no correspondence constraint is imposed, then the strategy for finding constrained equivalent forms of the components is the same as that for finding unconstrained equivalent forms: the constrained equivalent forms of the components are found separately and independently recombined. When a correspondence constraint is imposed, the constrained equivalent forms of the components can not be generated separately. Instead, to find additional equivalent forms of the original expression, the constrained equivalent forms of the components are found by manipulating the components identically; the expressions resulting from these manipulations are then used to replace the components in the generating expression.

To illustrate this process, consider the task of finding the constrained equivalent forms of the 4-point, short-time Fourier transform structure shown in Figure 6-8-a. Some of these equivalent forms will be found by manipulating the input sequences to the system BANK-OF-SEQUENCES in synchrony: thus, the set of sequences shown in Figure 6-8-b are considered simultaneously. Under this parallel manipulation, each of the equivalent-form transformations which could be applicable to the individual expressions is considered. Only the equivalent-form transformations which are applicable to *all* these parallel sequences will be completed and these transformations will be done on all of the sequences, simultaneously. The point of manipulation will progress inward by considering the inputs to the constrained expressions. Thus, the set of sequences shown in Figure 6-8-c is examined in parallel: this set of parallel sequences is formed by combining the correspondence constraint imposed on the inputs to the system BANK-OF-SEQUENCES with the correspondence constraint imposed on the inputs to the SEQUENCE-ADD and SEQUENCE-SUBTRACT systems. Without this second, additional constraint, two independent sets of parallel sequences would be manipulated, separately composed of the first and the second inputs to the addition/subtraction systems.

The point of manipulation progresses inward, so that the parallel sets of sequences shown in Figure 6-8-d and 6-8-e are also examined in turn. Pushing the correspondence constraint through the inner butterflies picks up another correspondence constraint, forcing all the inputs to be manipulated in synchrony. When the correspondence constraints are pushed inward from the position shown in Figure 6-8-e, all of the parallel sequences are collapsed into a single sequence, #<x>.

The answers from the inner constrained manipulations are propagated outward by combining the constrained equivalent forms with the operators which were dropped. Thus, any equivalent forms found for #<x> will travel outward by composing these forms with the dropped shift systems to create equivalent forms of the parallel expres-

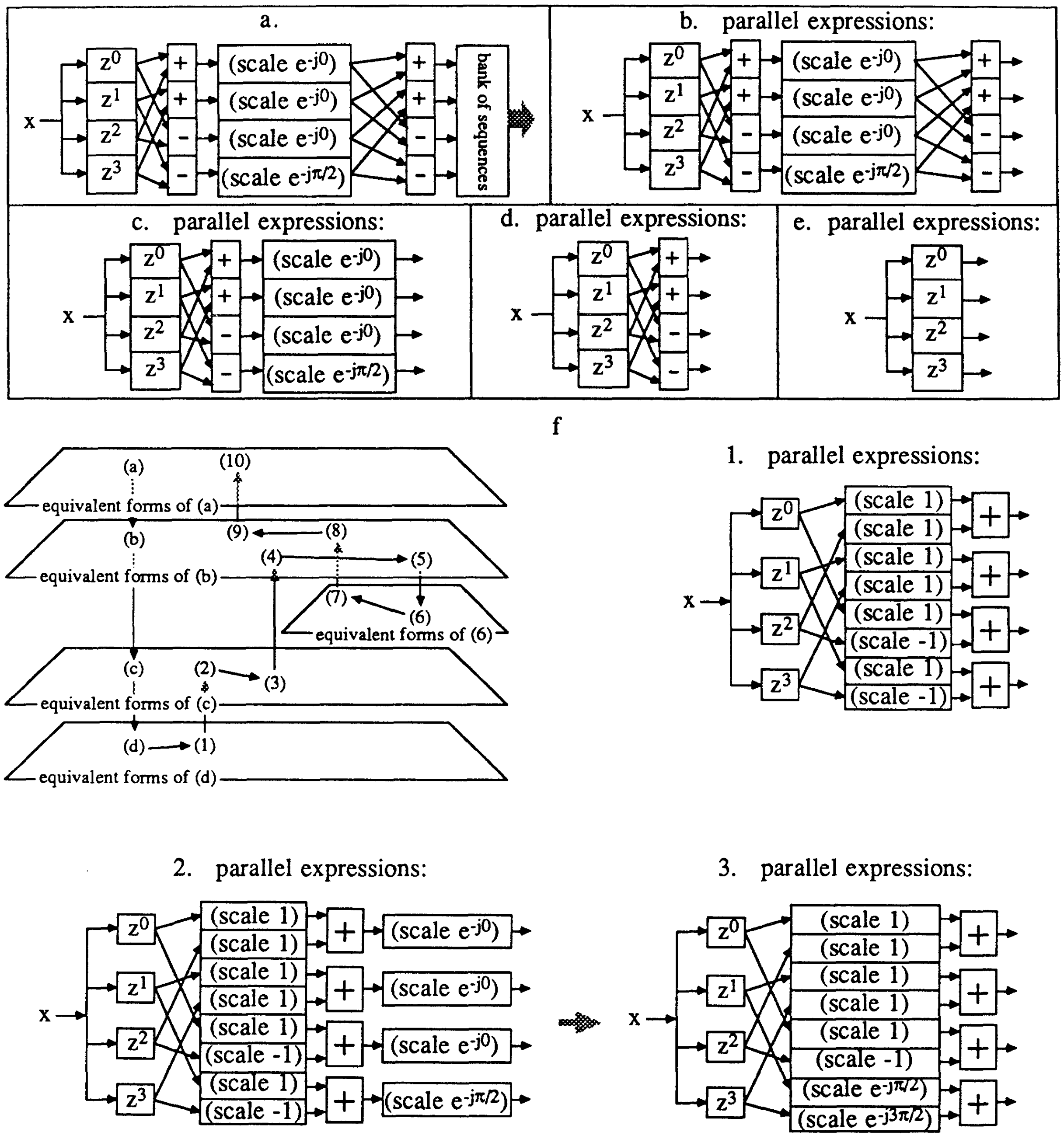
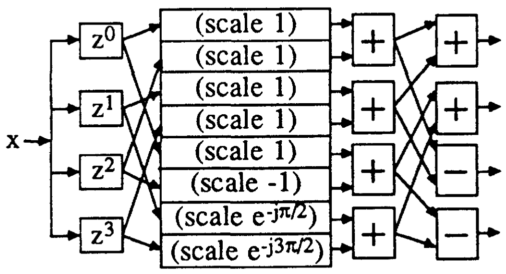


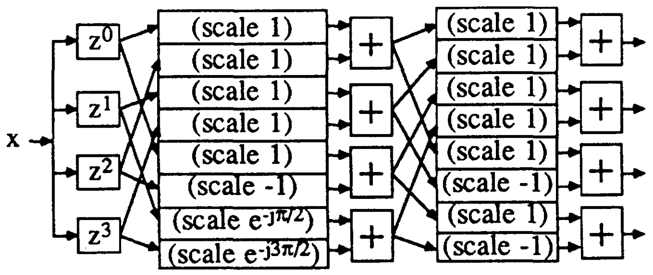
Figure 6-8: An example of parallel expressions and the effects of imposing regularity constraints

By imposing a correspondence constraint on an expression, the constrained manipulations of the inputs to that expression are forced to handle all the inputs in parallel. This correspondence constraint propogates inward through the subexpressions of the parallel inputs until a mismatch is detected. The effect of imposing correspondence constraints on the short-time Fourier transform structure is shown here. Part (a) shows the original constrained expression for the short-time Fourier transform. Parts (b) through (e) show the sets of subexpressions which must be manipulated in parallel. Part (f) sketches the progress of one line of manipulation.

4. parallel expressions:

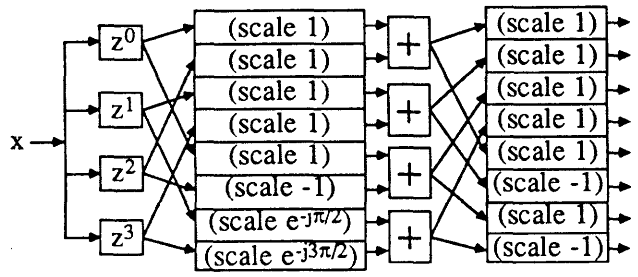


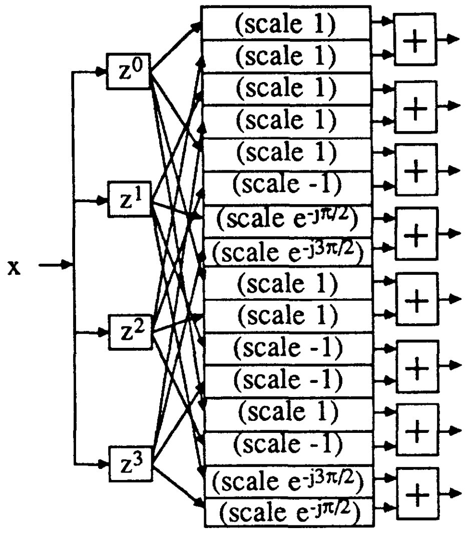
5. parallel expressions:



7. parallel expressions:

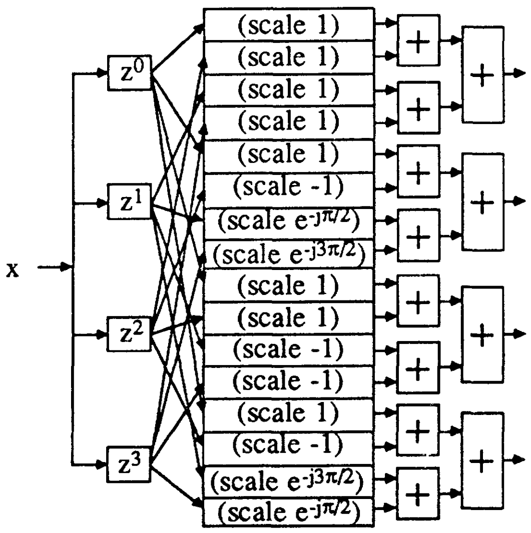
6. parallel expressions:

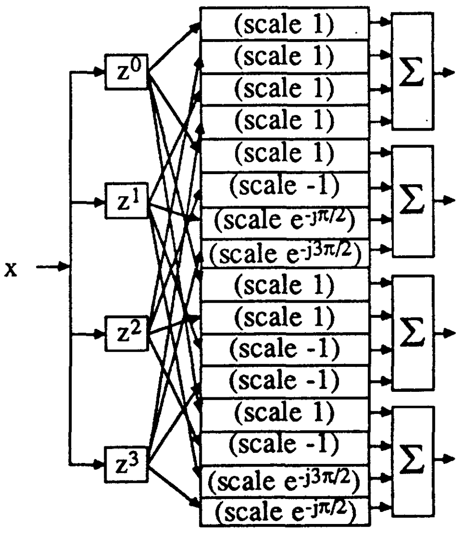




8. parallel expressions:

9. parallel expressions:





10. Direct-form implementation

sions shown in Figure 6-8-e. Each new set of parallel, constrained equivalent expressions is simplified using CONSTRAINED-SIMPLIFICATION and used to as a seed to find other constrained equivalent forms. This recursive process continues until no new, constrained equivalent forms are found. All of these sets of parallel equivalent forms will then travel outward by composing these parallel expressions with the addition and subtraction systems which were dropped in going from Figure 6-8-d to Figure 6-8-e. The cycle of constrained simplification and seeding of additional searches is repeated. The outward progress of constrained equivalent forms continues until all the equivalent forms of the original short-time Fourier transform are found. Figure 6-8-f sketches the progress of one line of manipulation.

As mentioned earlier, the actual transformation rules which are used to find constrained equivalent forms and constrained simplifications are the same rules as are used to find unconstrained equivalent forms and unconstrained simplifications, respectively. It is the manner in which these transformations are combined which provides the distinction between the constrained and unconstrained searches. In particular, in unconstrained searches, all subexpressions are manipulated independently while in constrained searches, parallel subexpressions are manipulated in synchrony.

6.3.2 Propagating correspondence constraints to a modified structure

When constrained expressions are manipulated to find transforms, constrained equivalent forms or constrained simplifications, new expressions are often generated on which the same correspondence constraints should be imposed. To illustrate, consider the manipulations shown in Figure 6-8-f: the result of these manipulations is a direct-form implementation of the short-time Fourier transform. In order to reflect the correspondence constraints of the original structure, this new bank of sequences should also include two correspondence constraints: the inputs to the bank of sequences should be constrained to coincide as should the inputs to the addition systems. Unless these constraints are imposed, this new form will introduce unconstrained structures into the constrained manipulations. ADE propagates structural constraints to new expressions automatically. Figure 6-10 provides two examples of this process. When a constrained expression is manipulated, the inputs which are constrained to be parallel are noted prior to manipulation. After each manipulation of a constrained structure, ADE will attempt to reimpose the noted correspondences on the modified structure.

To simplify this discussion of the propagation of constraints, Figure 6-9 introduces a way to describe the locations of the subexpressions in a signal-processing structure. The structure can be represented by a tree graph as shown in Figure 6-9-a: the parameters of a signal or the inputs to a system are the branches while the signal class or the system form the branch points. An occurrence of one of the original constrained inputs is found when a matching subtree is found. A match or mismatch is easily determined by simply comparing the two objects under consideration: since signals and systems have a unique representation, the input and the subtree will match if and only if they are the same object. Once a match is found, the location of that subtree can be represented by tracing out the path from the top of the complete tree to the subtree, as shown in Figure 6-9-b. Two or more subtrees in the modified structure can be forced to coincide only if they occur at the same depth in the overall tree. In addition to occurring at the same depth, the branch points along the paths to the two subtrees must all have the same branching factors: Figure 6-9-c shows an example of subtrees which occur at the same depth but which do not share corresponding branching factors on the paths leading to them. If the depth and the branching factors of the paths leading to the subtrees all match, a correspondence constraint can be imposed at the branch point where the paths part.

To reimpose the correspondence constraints from the original structure, coinciding subtrees are found in the modified structure. ADE will first search through the modified structure for coinciding subtrees, representing the original, parallel inputs. If subtrees representing the original, parallel expressions are found and their paths match both in depth and in branching factors, then a correspondence constraint is placed at the branch

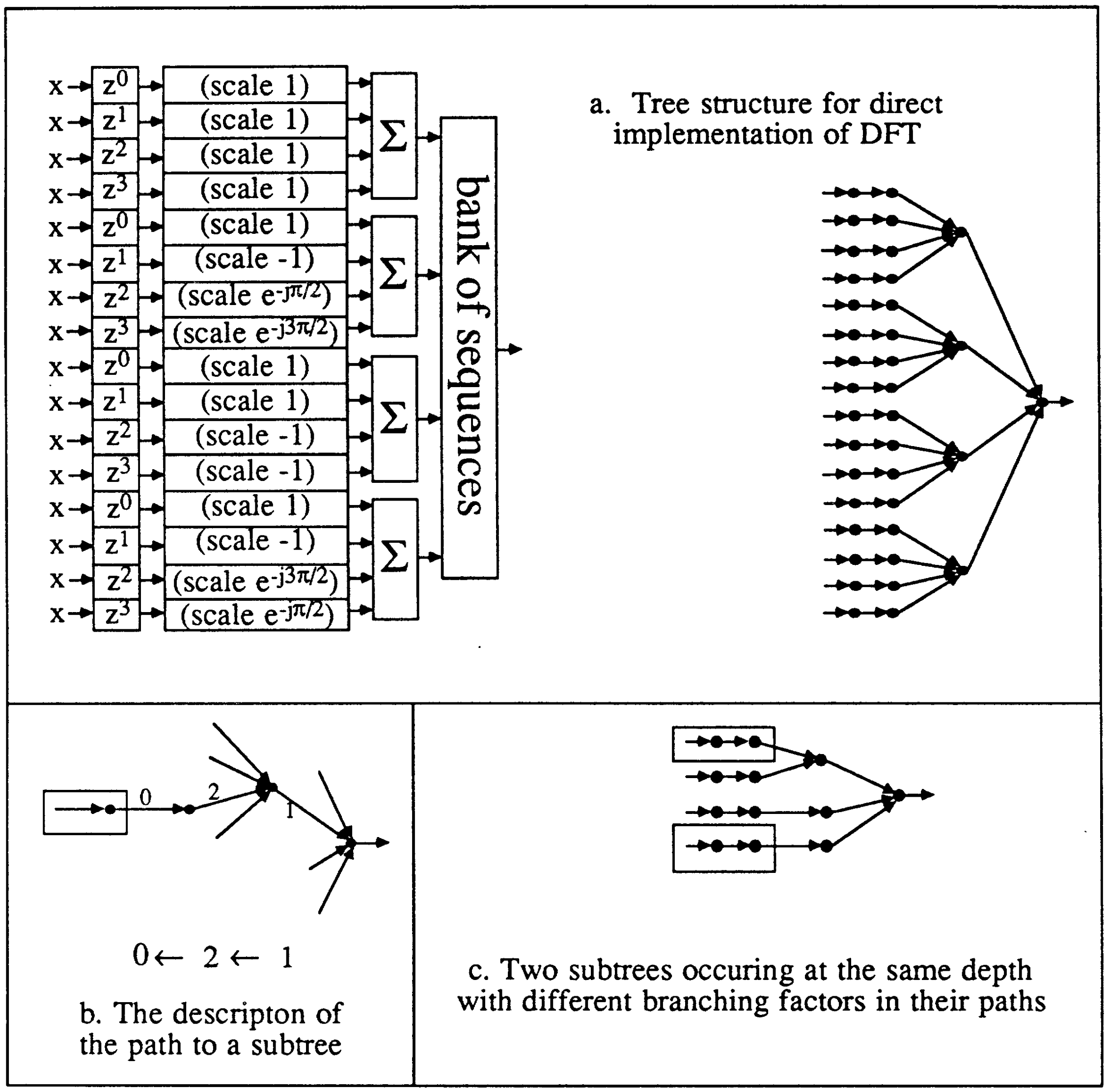

x
z0
z1
z2
z3
(scale 1)
(scale -1)
(scale e-jπ/2)
(scale e-j3π/2)
Σ
bank of sequences
a. Tree structure for direct implementation of DFT
0
2
1
0← 2← 1
b. The descripton of the path to a subtree
c. Two subtrees occuring at the same depth with different branching factors in their paths

Figure 6-9: Tree graph representations of signals

A signal processing expression can be represented by a tree graph: the parameters of a signal class or the inputs to a system are branches while the signal class or the system form the branch points. The locations of subtrees can be represented by describing the paths from the top of the complete tree to the subtrees. Two or more subtrees can be forced to coincide only if they occur at the same depth in the overall tree. In addition to occuring at the same depth, the branch points along the paths to the two subtrees must all have the same branching factors. If the depth and the branching factors of the paths leading to the subtrees all match, the subtrees can be forced to coincide by imposing a correspondence constraint at the branch point where the paths part.

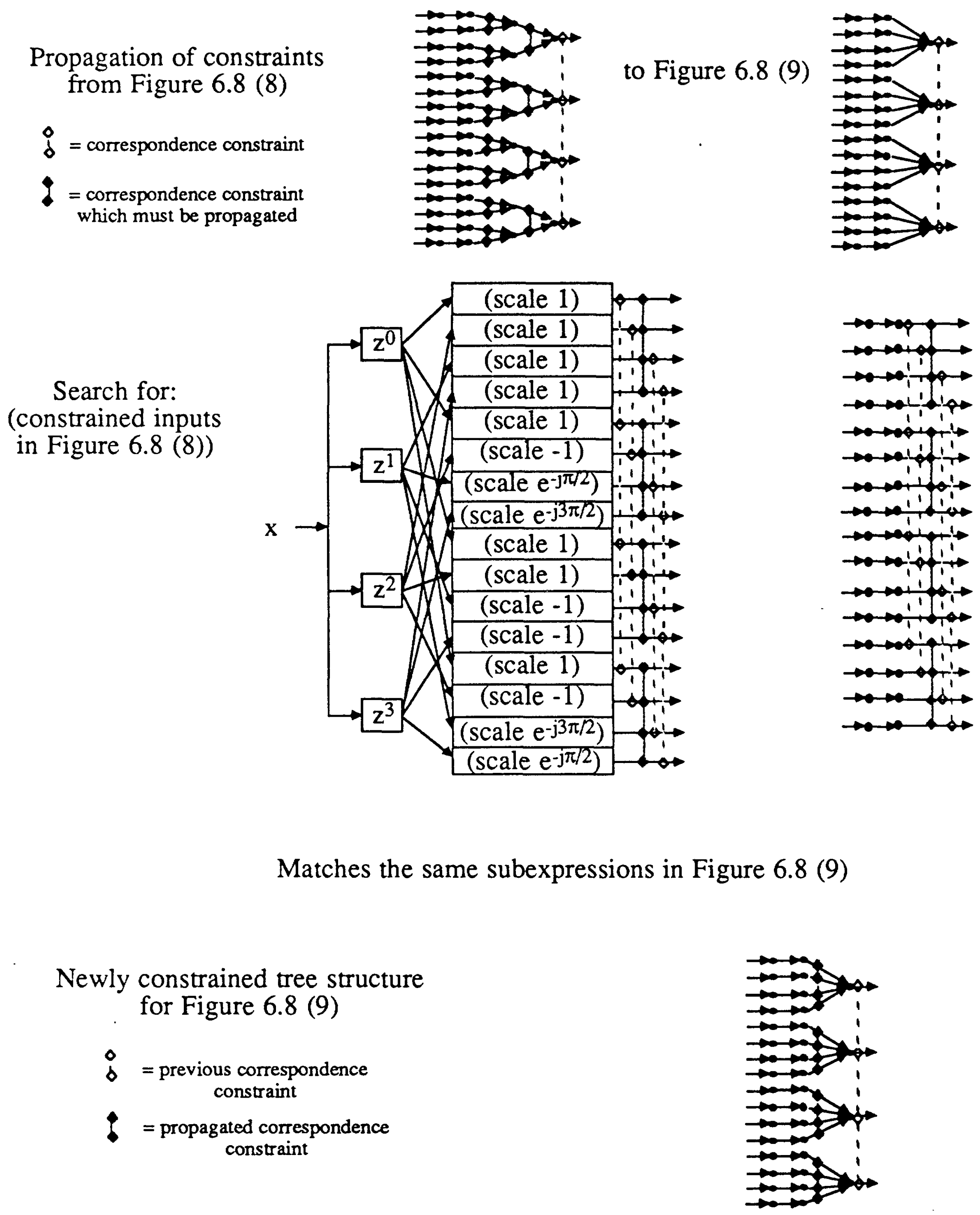
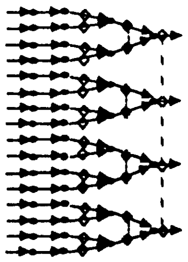

Propagation of constraints from Figure 6.8 (8)
= correspondence constraint
= correspondence constraint which must be propagated
to Figure 6.8 (9)
Search for: (constrained inputs in Figure 6.8 (8))
x
z^0
z^1
z^2
z^3
(scale 1)
(scale 1)
(scale 1)
(scale 1)
(scale 1)
(scale -1)
(scale $e^{-j\pi/2}$)
(scale $e^{-j3\pi/2}$)
(scale 1)
(scale 1)
(scale -1)
(scale -1)
(scale 1)
(scale -1)
(scale $e^{-j3\pi/2}$)
(scale $e^{-j\pi/2}$)
Matches the same subexpressions in Figure 6.8 (9)
Newly constrained tree structure for Figure 6.8 (9)
= previous correspondence constraint
= propagated correspondence constraint

Figure 6-10: Propagating correspondence constraints to modified structures

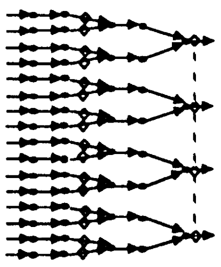
Propagation of constraints from Figure 6.8 (4)

= correspondence constraint

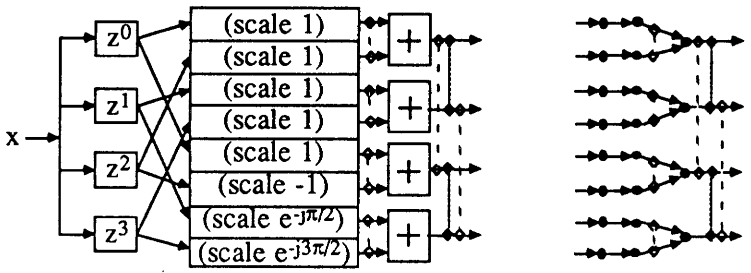
= correspondence constraint which must be propagated



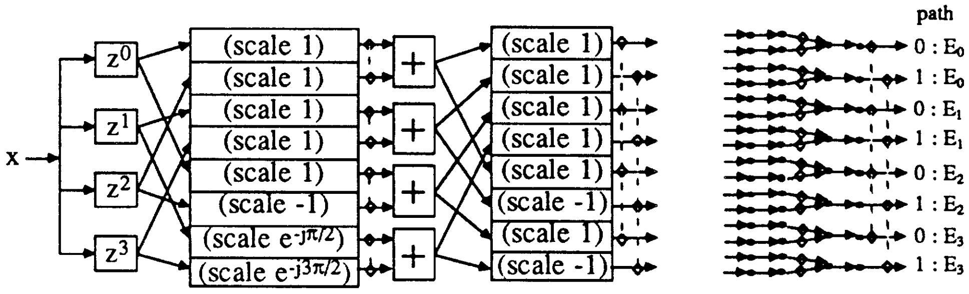
to Figure 6.8 (5)



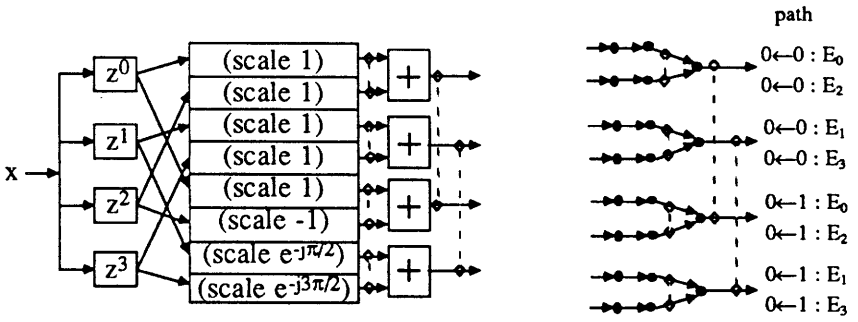
Search for: (constrained inputs in Figure 6.8 (4))



Doesn't match: (subexpressions of Figure 6.8 (5))



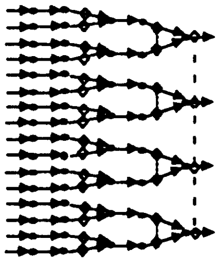
Matches: (subexpressions of Figure 6.8 (5))



Newly constrained tree structure for Figure 6.8 (5)

= previous correspondence constraint

= propagated correspondence constraint



point where the paths part. Otherwise, the modified structure will be searched for subtrees representing the components of the original set of parallel expressions. In this case, one of two approaches is used: when the parallel expressions themselves impose correspondence constraints on their inputs, all the component expressions are grouped into one set of parallel expressions; otherwise, multiple sets of parallel expressions will be created. By this process, ADE is able to propagate correspondence constraints from an original, regular structure to a modified structure, with minimal sensitivity to the addition or removal of uniform layers of processing.

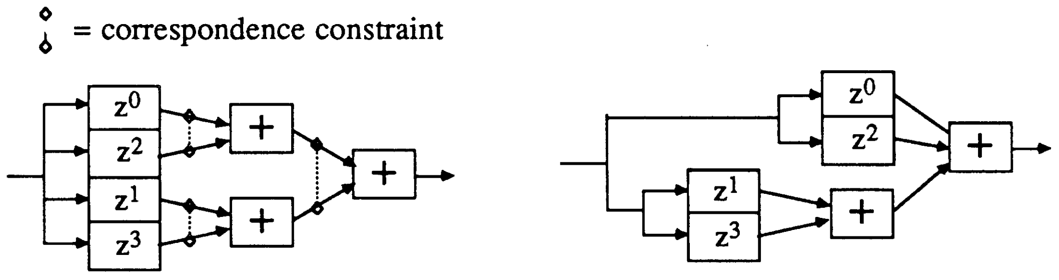
6.3.3 Manipulation of a single constrained expression

This section has discussed the propagation of correspondence constraints to modified expressions generated in searches for transforms and in constrained searches for equivalent expressions. The underlying assumption throughout this discussion has been that the intervening manipulations have not destroyed the regularity of the expression. The previous section outlined a process by which parallel expressions are manipulated in synchrony to maintain the correspondence between these expressions. However, the regularity of an expression can also be destroyed in manipulations of the signals or systems which represent the branch points originating the correspondence constraints. To illustrate, consider the rule shown on lines 6–12 of Figure 6-11-a: this rule collapses two cascaded applications of a commutative, associative operator into one application. In a search for the constrained equivalent forms of the structure shown in Figure 6-11-b, this pattern will match the outer constrained SEQUENCE-ADD output. If the rule were applied, the resulting modified structure, shown in Figure 6-11-c, would not exhibit the same internal regularity as the original structure: whereas the shifted inputs of the original structure all coincided, these inputs do not to coincide in the modified structure due the unilateral removal of one of the SEQUENCE-ADD systems. It is the manipulation of the expression which imposes the regularity constraint which introduces the irregularity.

This source of irregularity is easily removed by simply flagging the rules which in-

a.

```
(DEFINE-ABSTRACT-SYSTEM-CLASS (COMMUTATIVE-ASSOCIATIVE-SYSTEM *) *
        ; accept any parameters or inputs
      (COMMUTATIVE-SYSTEM ASSOCIATIVE-SYSTEM)     ; a subclass of these classes
  ("a commutative, associative system")
  ("a commutative, associative system output"
    (GOAL EQUIVALENT-FORM      ; collapse cascaded applications for an equivalent form
      :NAME SELF-APPLICATION
      :ALLOW-MAPOVER-MATCHES NIL      ; don't used on constrained branch points
      :GOAL-OBJECT (OUTPUT-OF ?SYSTEM (OUTPUT-OF ?SYSTEM &REST ?INPUTS)
                                  &REST ?OTHER-INPUTS)
         ; one of the inputs is an output from same system
      :ANSWER (APPLY 'OUTPUT-OF SYSTEM (APPEND INPUTS OTHER-INPUTS)))
    ... ))
```



b. c.

Figure 6-11: The introduction of irregularities within a single, constrained expression due to dissimilar modification of the coinciding inputs

Irregularities can be introduced into a constrained structure by manipulations involving the actual expressions which impose the constraint. The rule for collapsing two cascaded applications of a single commutative, associative operator into one application, shown in part (a) provides an example of this effect. When applied to the outer constrained SEQUENCE-ADD of the structure shown in part (b), it introduces a mismatch between the operators inputs, as seen in part (c).

troduce irregularity when applied to the branch-point expression of a correspondence constraint. The additional effort required when programming the transformation rules is negligible compared to the processing which would be required to automate this process: of the rules currently included in ADE, fewer than 20 in 850 are flagged as sources of irregularity. With the provisions described in this section, the modified structures resulting from searches for transforms and searches for constrained equivalent expressions will exhibit the same general internal regularity as the original expressions. As a result of these provisions, ADE is able to propagate the given correspondence constraints to the related modified structures.

6.4 Summary

This chapter has discussed the representation and propagation of correspondence constraints. Correspondence constraints can either be explicitly pointed out by the user or can be included in the description of the equivalent forms of a high-level signal processing operation. For example, the regularity constraints described in section 6.2 would be included in the description of this equivalent of the short-time Fourier transform.

In ADE, MAP-OVER is used to impose correspondence constraints. Once a correspondence constraint is imposed, ADE propagates that constraint inward through the computational structure until a mismatch is detected. Manipulations which may introduce irregularities into a constrained expression are manually flagged, allowing these sources of irregularity to be avoided.

Use of these constraints allows the space searched for equivalent forms and efficient implementations to be drastically reduced. Using correspondence constraints, the size of search space is reduced from $\mathcal{O}(M^N)$ to $\mathcal{O}(M)$ where N is the number of the parallel subexpressions and M is the number of equivalent forms which can be found for each of these parallel subexpressions. This search space reduction was illustrated dramatically with the FSK-code detector in Chapter 3: regularity constraints reduced the design space of the 16-channel modulated filter bank by factors between 10^{18} and 10^{58}, depending on the impulse response used in the filter bank.

This approach is a heuristic pruning strategy: there can be no assurance that the uncovered implementations actually include the most computationally efficient implementations.[1] However, this approach exploits the internal structure of the expression being manipulated to reduce the search space. The regularity of the algorithm is due to the similarity between separate parts of the expression: correspondence constraints are imposed on similar expressions which travel through similar processing layers. Forcing this regularity to be maintained in the equivalent expressions is a reasonable way to avoid the combinatorial growth of the design space: since the coinciding expressions are similar, an implementation which is efficient for one expression will usually be efficient for the other coinciding expressions.

This justification for regularity constraints is supported by revisiting the problem of non-integer sampling rate conversion. One of the applications of non-integer sampling rate conversion is in displaying a film sequence on television: film is shot at a rate of 24 frames per second while television displays at a rate of 60 fields per second, where adjacent fields are interlaced to create a frame rate of 30 frames per second. This rate conversion problem can be dealt with as a one-dimensional problem using a 4:5 temporal sampling rate conversion.[2] The results of a constrained search for the efficient implementations of this rate conversion included the polyphase matrix structure shown in Figure 6-12. This structure has the same structure as the implementation reported by Myers (1986) (see Figure 2-5-b).

[1]In fact, often, there are a large number of computationally efficient implementations which will not be found, when regularity constraints are enforced: if the equivalent subexpressions of two computationally efficient implementations can be intermixed without reducing the overall efficiency, then the resulting structure will be another computationally efficient implementation. One possible way of obtaining these structures while still using regularity constraints to reduce the size of the search space is discussed in Chapter 8.

[2]With this approach, each film frame would be digitized on the combined rasters of the interlaced fields and then separated into two interlaced sequences of fields. These digitized film fields would then be used as inputs to the two one-dimensional 4:5 sampling rate conversions, one creating the even fields and the other, the odd fields. The final television sequence would then generated by interleaving these two sets of fields.

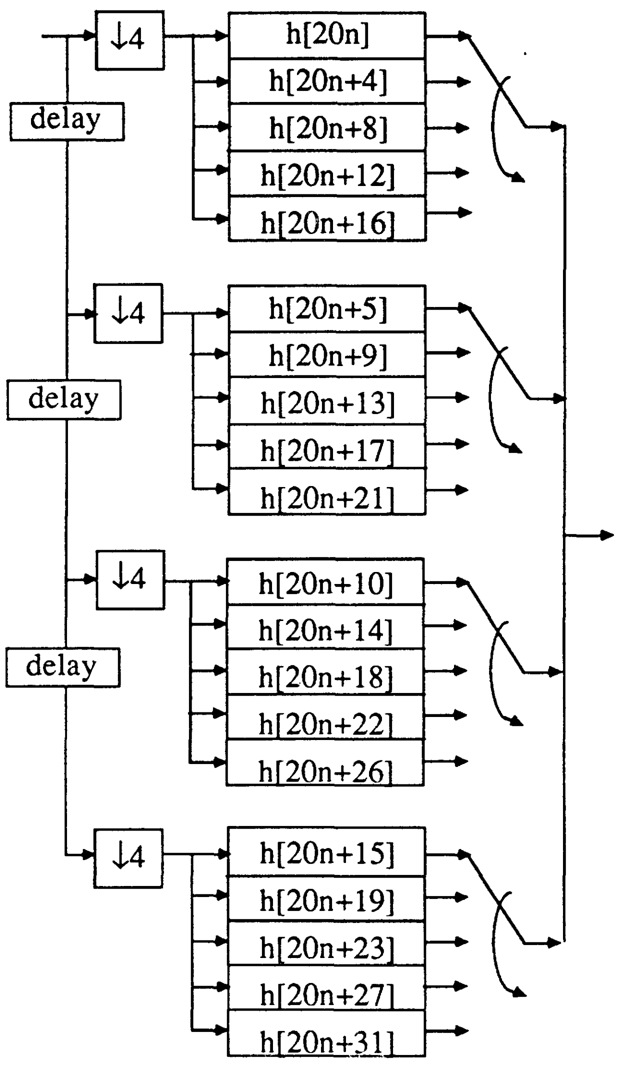


Figure 6-12: The polyphase matrix structure for a 4:5 conversion ratio

Using parallel manipulations on a 4:5 non-integer sampling rate conversion, this polyphase matrix structure was discovered. This implementation has the same structure as the 2:3 polyphase matrix reported by Myers (1986).

The discovery by ADE of this polyphase matrix structure using constrained manipulations suggests that maintaining correspondence constraints results in a reasonable sampling of the full solution space: all the same general implementations are still uncovered, even though the full solution space is not considered.

Chapter 7

Cost Measures

This chapter develops a framework for ranking implementations using cost structures. The two driving motivations throughout this development are to accurately reflect the relative costs of alternate implementations and to remove as many of the superfluous implementations as possible without removing any implementations which would actually be of interest to the user. The first section describes the external characteristics and the interactions of the cost measures. The second section discusses the internal propagation of costs: this propagation is necessary to accurately estimate the relative cost of different implementations.

7.1 External Characteristics of Signal-Processing Cost Measures

This section examines both a metric space which seems appropriate for signal processing and some of the common distributions of cost versus time that arise in signal processing.

7.1.1 Cost metric space

The most obvious, and often the most appropriate, cost measure for signal-processing algorithms is a measure of the computational requirements of the implementation.[1] The way to measure the computational requirements of an algorithm is less obvious. Until the architecture on which the algorithms are to be run is selected, no single number can be used to compare the execution times of equivalent implementations: the relative cost of real additions, complex multiplications and memory references, for example, will vary depending on the identity and configuration of the host computer as well as the particular low-level implementation chosen for the operations. Therefore, a vector of costs is necessary to characterize each implementation. Operation counts and memory requirements are included in this vector. By allowing the cost to remain in terms of its individual components, other unrelated measures of cost, such as a rudimentary sensitivity analysis, can be easily incorporated.

Using a vector of costs for each equivalent implementation, a set of "undominated" implementations can be determined, where an "undominated" implementation is one for which there is no other equivalent implementation with a strictly smaller cost measure. In this context, a cost vector is said to be strictly less than another cost vector when the first vector is always component-wise less than or equal to the second vector and when the first vector has at least one component that is smaller than the corresponding component of the second vector. To illustrate, consider three implementations: implementation a, requiring 2 complex additions and 4 real additions; implementation b, requiring 5 complex additions and 2 real additions; and implementation c, requiring 7 complex additions and 3 real additions. Implementations a and b would be undominated implementations. Implementation c is dominated by implementation b but not by implementation a. For two-dimensional vectors, removing dominated implementations is graphically equivalent to removing all the implementations whose cost vectors are in the first quadrant of a coordinate system referenced from the cost vector of another implementation (Figure 7-

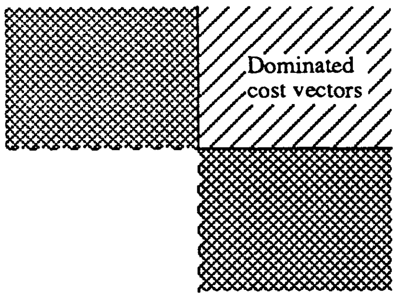
[1]Another cost measure that would often be useful is a stability/sensitivity analysis.

1). By extending the boundary lines, it becomes obvious that, if an implementation is undominated, then no other implementation can fall in the third quadrant in its cost space: otherwise, the first implementation would not be undominated. Thus, all other undominated implementations must lie in the second or fourth quadrants. This graphical device will allow us to compare the efficiency of domination by partial ordering with the efficiency of domination using added bounding constraints.

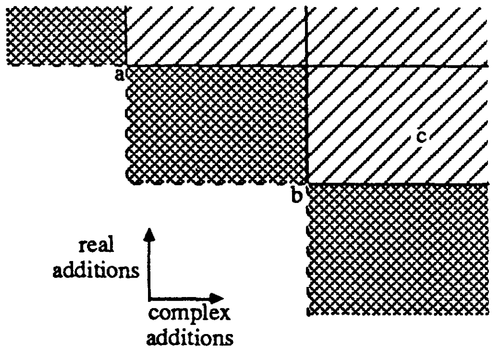
As discussed above, the relative costs of different operations can not be determined until the hardware capabilities are known. However, upper and lower bounds on the relative costs of different operations can often be determined, independent of the architecture. These bounding equations can be used to further reduce the number of implementations presented to the user. Some limiting relations that ADE uses are: complex multiplication costs at least as much as complex addition; real multiplication costs at least as much as real addition; complex addition costs at least as much as real addition but no more than two real additions; and complex multiplication costs at least as much as three real multiplications but no more than six real multiplications.[2] Graphically, the additional constraints increase the percentage of the vector space which each implementation removes from consideration. Continuing with the previous example, these additional bounds on the relative costs of real and complex additions enlarge the dominated region from being the first quadrant to being the shape shown in Figure 7-2-a. Again, by extending the boundary lines of the domination mask, the areas in which undominated implementations can fall, given the location of one undominated implementation, is easily seen. Using these two simple constraints, the percentage of the cost plane in which a second undominated implementation can fall is reduced from 50% to just over 10%.

Thus, by using a vector of operation counts for each implementation, the computational requirements of each implementation can be compared and a partial ordering can be completed. Simple domination between implementations results directly from

[2]A complex multiplication of $(a_r + ja_i)$ and $(b_r + jb_i)$ is comparable to three real multiplications using $(c_1 - c_2) + j(c_3 - c_1 - c_2)$ where $c_1 = a_r b_r$, $c_2 = a_i b_i$ and $c_3 = (a_r + a_i)(b_r + b_i)$ and ignoring the cost of the additions. It is comparable to six real multiplications using $(a_r b_r - a_i b_i) + j(a_r b_i + a_i b_r)$ and counting the cost of the real additions as the same as a real multiplication.

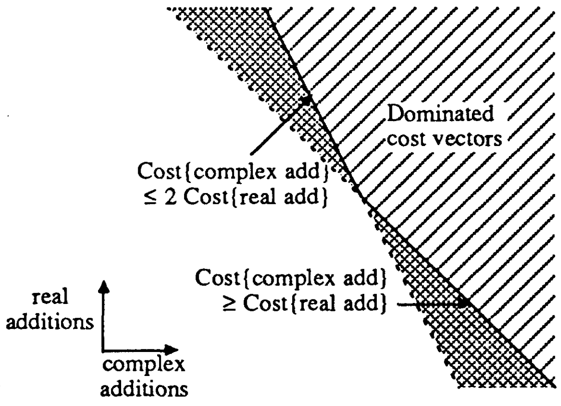


a) Using only a partial ordering of the cost measures by component-wise comparison, some implementations can be eliminated, leaving only undominated implementations. For a two-dimensional cost vector, given the location of one undominated implementation, any other undominated implementation must lie in either the second or the fourth quadrant of the coordinate system referenced from the undominated cost vector.

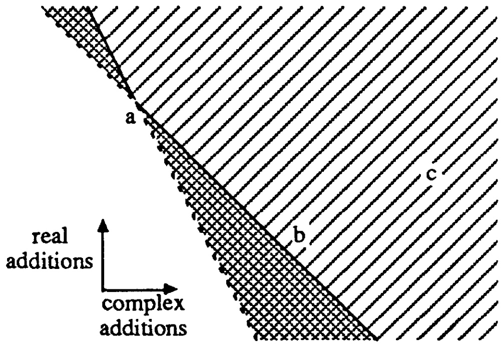


b) Using only a partial ordering of the cost measures of the example developed in the text, implementation c is dominated by implementation b and implementations a and b are undominated.

Figure 7-1: Domination using partial orderings of computational requirements



a) By adding limiting constraints on the relative costs of related operations, the dominated area provided by each implementation is increased. Given the location of one undominated implementation, the area in which any other undominated implementation must lie is corresponding decreased. This idea is illustrated here for cost vectors of real and complex additions.



b) Adding two bounding constraints on the relative costs of complex additions and real additions to the example developed in the text, implementations b and c are dominated by implementation a.

Figure 7-2: Domination using limiting constraints on the relative costs of related operations.

this partial ordering. Additional comparisons can be made between implementations by including limiting relations between the various operations.

7.1.2 Time distribution of cost

In order to adequately characterize the computational costs of a signal, the distribution of cost versus the signal index must be described. To illustrate, consider the cost of taking an N-point FFT of a discrete-time sequence: the cost of computing samples outside the indexing interval $[0\ N]$ is zero, since these samples are known to be identically zero, while the cost of computing samples inside the interval is the cost of computing the samples of the input sequence plus the classic $N \log_2 N$ complex additions and $\frac{N}{2} \log_2(\frac{N}{2})$ complex multiplications. Simply giving the $\mathcal{O}(N \log_2 N)$ dependence does not adequately characterize the cost.

The FFT example also illustrates another type of time dependence in computational costs, namely the association of costs with blocks of samples. To be more explicit, a request for a single sample of the FFT in the indexing interval $[0\ N]$ incurs the same computational cost as requesting all the samples in that interval. Thus, the costs in the interval $[0\ N]$ are associated with all N samples and requesting one sample in an indexing block is computationally equivalent to requesting all the samples. This stands in sharp contrast with the behavior of operations like sequence addition or multiplication. With these operations, the computational costs of distinct samples are independent: the indexing block size is one sample.

This development results in a cost structure being associated with each implementation. Each cost structure consists of "cost intervals", where a cost interval describes an interval with uniform cost characteristics. Each cost interval indicates the indexing interval which it describes; the computational block size within that interval; and the cost vector incurred by each indexing block. As illustrated in Figure 7-3, these cost structures are able to characterize many of the different time dependencies that arise in signal processing in a simple, efficient manner.

a) Sequence multiplication (assuming the input samples are cost-free): The cost of sequence multiplication is uniform across the non-zero support of the output sequence, with the cost of each sample being independent of the costs for any other sample. Therefore, in this example, there is only one cost interval with an indexing block size of one sample and a cost vector for each block of one complex multiplication.

b) 16-point FFT (assuming the input samples are cost-free): Samples of the FFT that lie outside its non-zero support are cost free. Within the non-zero support, the FFT uses block processing to achieve a cost for computing the full non-zero support of N log N complex additions and N/2 log (N/2) complex multiplications. Therefore, there are three cost intervals. The cost intervals covering below 0 and above 15 have an indexing block size of 1 sample and a null cost vector for each block. The cost interval covering [0 16] has an indexing block size of 16 samples and a cost vector for each block of 64 complex additions and 24 complex multiplications.

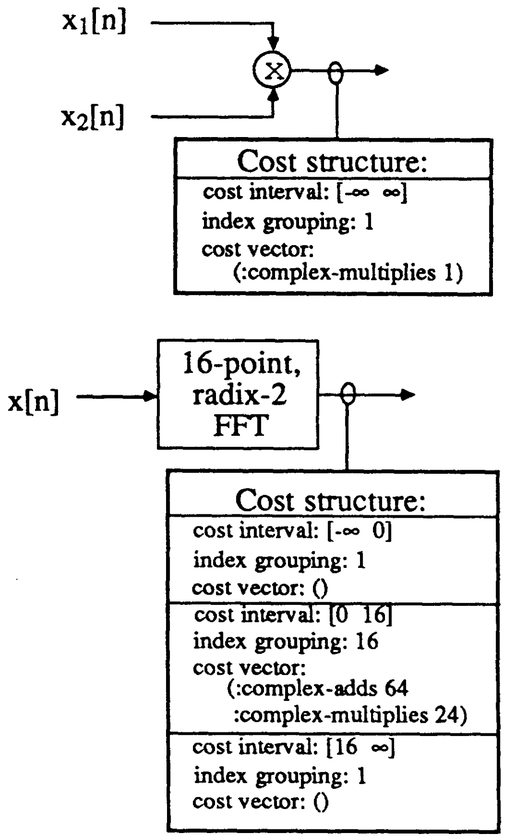


Figure 7-3: Examples of cost structures incorporating time dependencies

Unfortunately, domination of one implementation over another is greatly complicated by this explicit distribution over time. Although the average cost per sample could be used in determining dominance, a variety of situations arise where only some of the sample values of the implementations are required, making this an invalid measure of the actual cost. For example, if a convolution is followed by windowing, comparison of the average costs of block convolution and direct-form convolution could lead to exactly the opposite conclusion from that which is correct (Figure 7-4). Thus, dominance of one implementation over another can only be asserted when the cost vector of every interval of the dominating implementation dominates the corresponding cost vector for

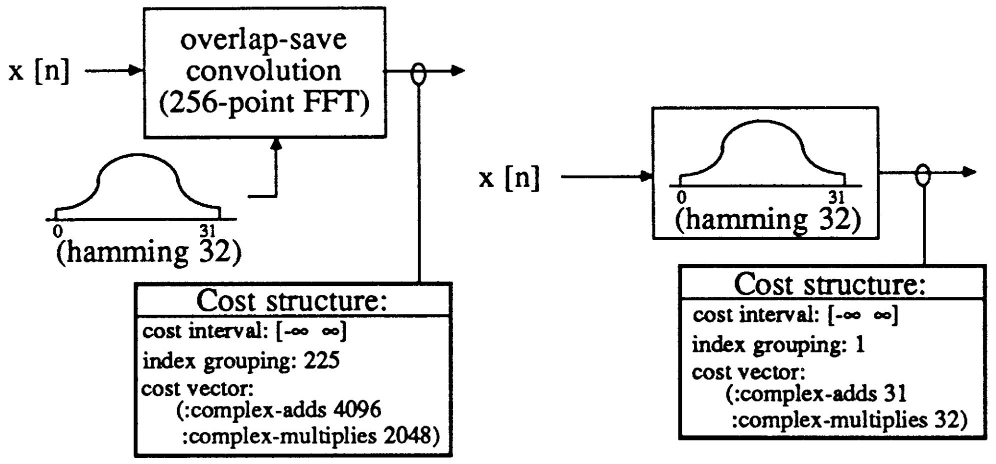


Figure 7-4: Comparison of cost structures with unequal indexing block sizes
Dominance of one cost vector over another when the computational block sizes are unequal is difficult to establish, since the number and distribution of samples that will be computed is not generally known. Consider the two implementations of a convolution shown here. Using the average cost per sample, the overlap-save requires about a third the number of complex multiplications and half the number of complex additions. However, if the output of the convolution is subsequently windowed so that 64 or fewer samples are retained, the direct-form implementation is more efficient than the overlap-save.

the dominated implementation and, if the indexing block sizes are unequal, when the cost per block for the larger indexing block dominates the cost per block for the smaller indexing block.

To summarize this discussion of signal-processing cost measures, basic operation counts are used as a measure of the cost of an implementation. EFFICIENT-IMPLEMENTATIONS and CONSTRAINED-EFFICIENT-IMPLEMENTATIONS, only present implementations which are undominated. An implementation that dominates an alternate implementation must have a cost structure which dominates the cost structure of the alternate implementation over all cost intervals. Dominance on a cost interval can only be established in two cases: if the indexing block sizes are the same, then dominance is established by dominance of one cost vector over the other; if the indexing block sizes are unequal, then dominance can only be established if the cost per block for the larger indexing block is smaller than the cost per block for the smaller indexing block.

7.2 Internal Behavior of Signal-Processing Cost Measures

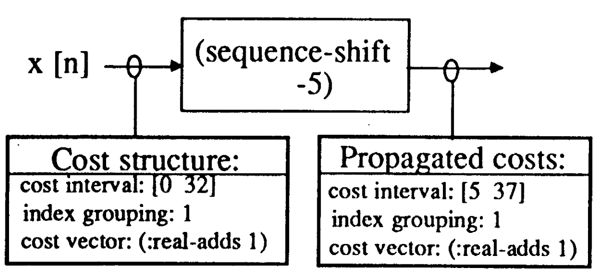
On all but the most trivial signal-processing systems, computation of the output signal requires computation of some or all of the input signals, since the output sample values depend on the sample values of the input signals. Thus, computation of the output signal directly incurs the local cost of the system and indirectly incurs the cost of associated with computing the required samples of the input signal. This section discusses some of the issues that arise in reflecting these indirect costs.

7.2.1 Propagation of cost

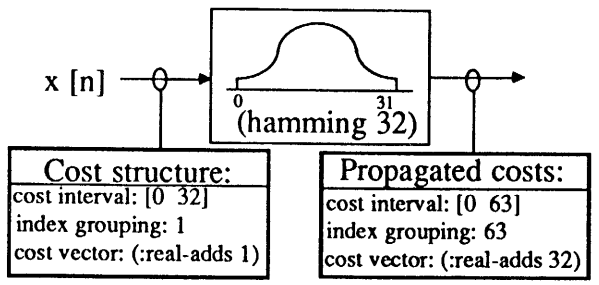
To be able to reflect the cost of computing a system input in the output cost, the time dependence of the output signal on the samples of each input signal and both the size and the position of the computational blocking must be available. Some simple examples to illustrate the necessity of this knowledge are provided in Figure 7-5. For the sake of clarity, this figure only shows the propagated costs: the local costs of the systems are omitted.

One approach to describing input/output relations is discussed here. Using this approach, the output dependence on each input is described by a set of "propagation intervals". A propagation interval describes the propagation characteristics of the system on a particular output indexing interval, thus allowing time-varying input/output dependencies. In addition to the interval over which the dependence applies, the propagation interval contains: an output interval representing one computational block;[3] the corresponding input interval on which the output interval depends; and the "sense" of

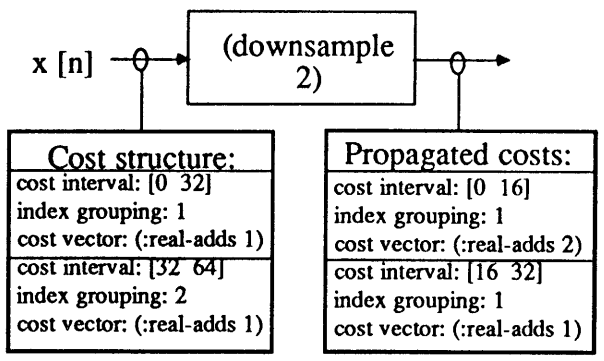
[3]The prototypical output block is not forced to fall inside the extent of the propagation interval, but rather simply to reflect the relative sizes and placements of corresponding input and output blocks.



The cost intervals for the input to a shift system must themselves be shifted. The shift system does not affect the index grouping of the input cost intervals.



For convolution, the full cost of computing all of the samples of the non-zero support of the input are accumulated into a single cost interval covering the non-zero support of the output, with an indexing block size equal to the length of the output non-zero support.



The cost intervals for the input to a downsampler must themselves be 'downsampled'. To downsample a cost interval:
- The associated indexing interval is downsampled.
- The downsampling rate is separated into a product of factors that evenly divide the indexing block size and of factors that do not evenly divide the block size. The output indexing block size is the input indexing block size divided by the first set of factors and the output cost vector is the input cost vector multiplied by the second set of factors.

Figure 7-5: Propagation of costs from system inputs to the system outputs

When the computation of a system output requires the computation of some of the system inputs, the cost of computing the system inputs should be reflected in the cost of the system output. To accurately reflect the time distribution of these indirect costs, knowledge of both the input/output sample dependencies and the local blocking size and position of the system's computation is required. Some examples are provided to illustrate a range of input/output dependencies and computation blocking: the effect of the system on the cost structures of the inputs is described for each of the systems. Only propagated input costs are shown: the local cost of the systems are omitted for clarity.

the input/output relation which is described below. Using this input/output block pair, the input dependency for a prototypical output block is given. Since each propagation interval typically spans more than one block, the input/output relations of the remaining blocks are determined by tessellating the output indexing interval using the output computation block and tessellating the domain of the input signal using the input block. Then, if the "sense" of the input/output relation is "normal", the output block immediately to the left of the prototypical output block is associated with the input block immediately to the left of the prototypical input block, and so on. If the "sense" of the input/output relation is reversed, the output block immediately to the left of the prototypical output block is associated with the input block immediately to the right of the prototypical input block, and so on. Figure 7-6 illustrates this propagation for the systems shown in Figure 7-5.

Obviously, this propagation is not as general as it could be. In particular, provisions could be made for allowing non-contiguous or overlapping input blocks. Non-contiguous input blocks would allow a more accurate representation of downsampling: this is an unfortunate limitation of the chosen representation. Overlapping input blocks are deliberately disallowed to avoid cost inflation. To demonstrate this possibility, Figure 7-7 shows three possible ways of propagating the input cost to the output cost for a causal IIR filter. The first two ways shown in Figure 7-7 do not require the use of overlapping input blocks, only the last way does. Unfortunately, the last way also includes the cost of computing the input sample at $x[n_0]$ in the cost of the output sample at $y[n_0]$ and in all subsequent output samples as well. So, overall, the cost of computing each input sample is included in the output cost an infinite number of times. Not allowing overlapping input blocks avoids the possibility of this type of cost inflation.

Once the dependencies of a system output on each of the system inputs is described, the input cost structures can be modified to conform with their time distribution as seen from the output. The algorithm used to do this is included in the environment of ADE.

As a result of including the costs of system inputs in the output cost, cost structures

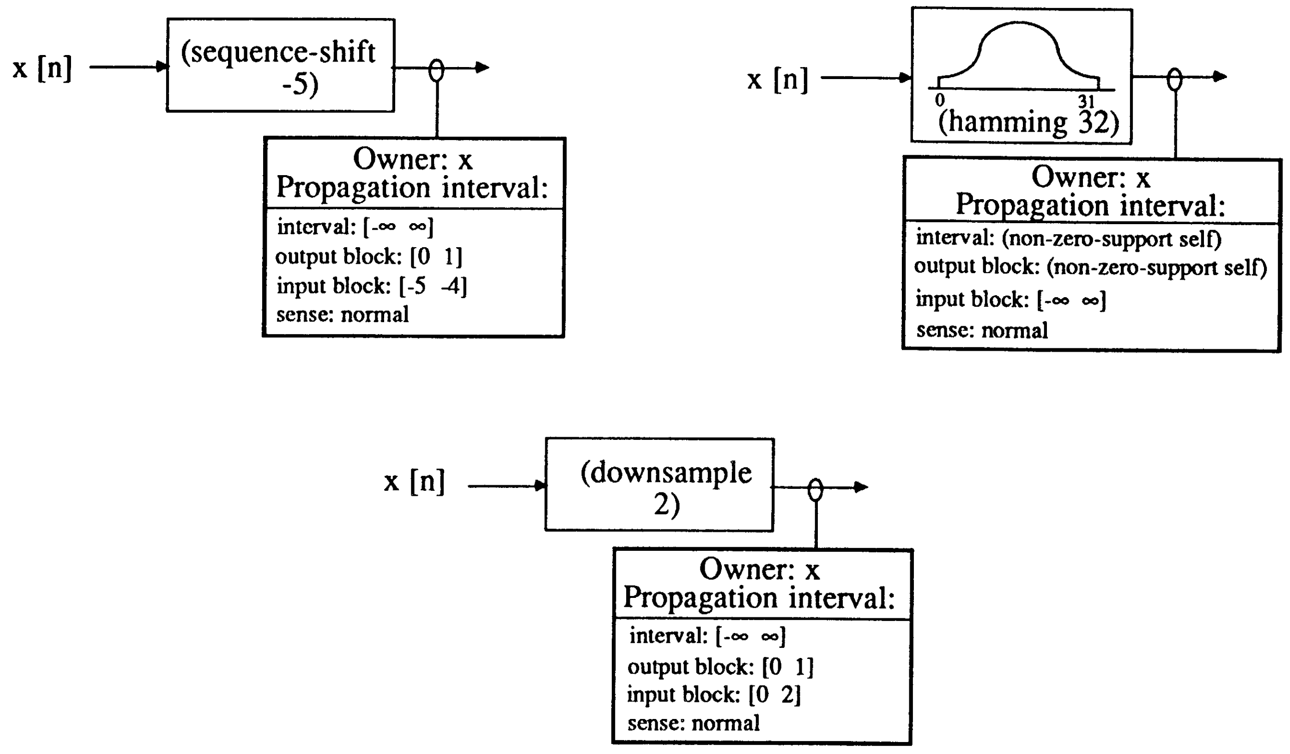
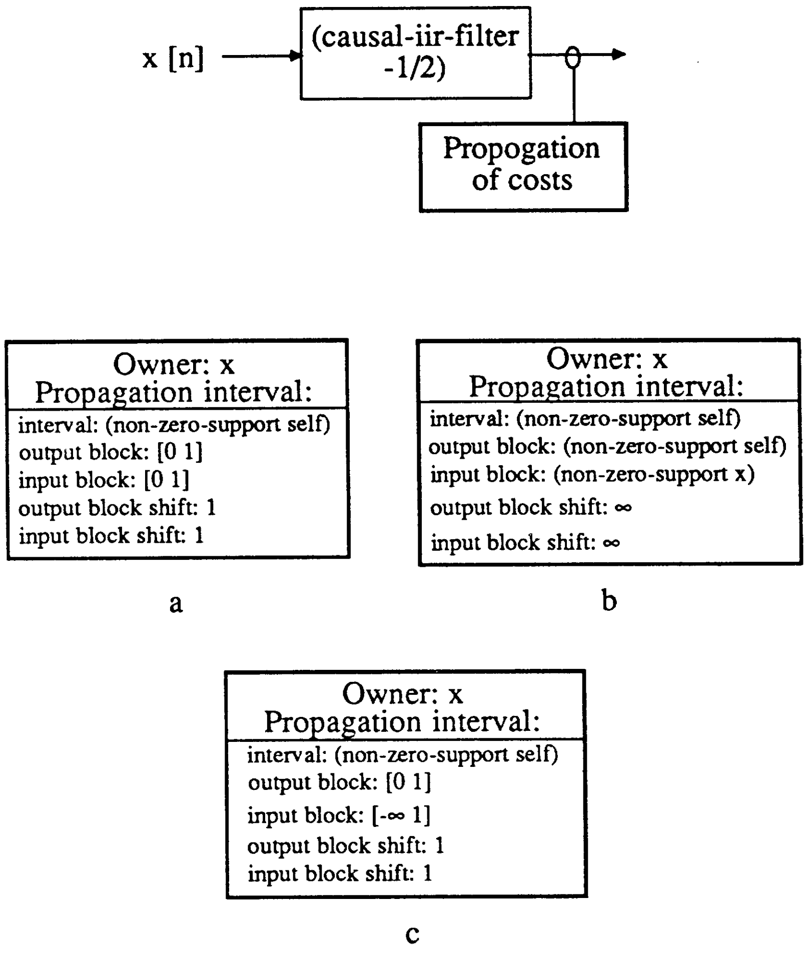


Figure 7-6: Input/output descriptions of the systems shown in Figure 7-5



a

Owner: x
Propagation interval:
interval: (non-zero-support self)
output block: [0 1]
input block: [0 1]
output block shift: 1
input block shift: 1

b

Owner: x
Propagation interval:
interval: (non-zero-support self)
output block: (non-zero-support self)
input block: (non-zero-support x)
output block shift: ∞
input block shift: ∞

c

Owner: x
Propagation interval:
interval: (non-zero-support self)
output block: [0 1]
input block: [-∞ 1]
output block shift: 1
input block shift: 1

Figure 7-7: Three alternate ways of propagating the input cost through an IIR filter

The proposed description of input/output relationships of systems is not as general as it could be. In particular, no provisions are made for allowing overlapping input blocks. This restriction is most noticeable in recursive computations, such as an IIR filter. If this restriction is removed, there are three alternate ways of naturally propagating costs through an IIR system. Unfortunately, the third, which in some ways reflects this cost most accurately, also results in cost inflation, by including the cost of each input sample in the output cost an infinite number of times.

will eventually need to be added. If the corresponding cost intervals in the two cost structures being added have the same indexing block size, the addition is straightforward: the cost vectors are simply added. When corresponding cost intervals do not have equal block sizes, the addition is more involved. One option in this case would be to use a block size that is the least common multiple of the two block sizes and add the cost vectors, after scaling appropriately (Figure 7-8-a). The primary difficulty with approach arises when one of the cost vectors has an infinite block size.

To illustrate, consider problem of describing the cost of computing the output from one of the modulated filters in the matched filter bank shown in Figure 3-1: that is $y[n] = x[n] * (e^{-j\frac{2\pi}{N}kn} r_N[n])$ where $r_N[n]$ is an N-point causal, rectangular window. Assuming that the sequences $x[n]$, $e^{-j\frac{2\pi}{N}kn}$ and $r_N[n]$ are all cost-free, the propagation and addition of costs using least-common-multiple block sizes is shown in Figure 7-8-b. The cost of computing $y[n]$ is the sum of: N multiplications and $N-1$ additions per sample, for the local computations in the convolution and N multiplications per ∞ samples, for the one-time cost of computing $e^{-j\frac{2\pi}{N}kn} r_N[n]$ (Figure 7-8-b). If least-common-multiple block sizes are used for adding up the costs, the cost of the output is ∞ multiplications and ∞ additions per ∞ samples. To highlight the problem with this, consider another way of getting the same output signal shown in Figure 7-8-c: that is, $y[n] = x[n] * h[n] - x[n] * h[n-N]$ where $h[n] = e^{-j\frac{2\pi}{N}kn} u[n]$. The cost of computing $y[n]$ in this manner is the sum of: 1 addition per sample, for the local computations in the sequence subtraction; ∞ multiplications and ∞ additions per sample, for the local computations in the convolution and ∞ multiplications per ∞ samples, for the one-time cost of computing $h[n] = e^{-j\frac{2\pi}{N}kn} u[n]$ (Figure 7-8-c). Using least-common-multiple block sizes for adding up the costs, the cost of the output is again ∞ multiplications and ∞ additions per ∞ samples. Obviously, the number of multiplications required for this second implementation is greater than the number of multiplications required for the first but the cost structures for the outputs are the same. Thus, if least-common-multiple block sizes are used in adding cost structures, the ability of the resultant cost structures

Cost structure:
cost interval: [-6 0]
index grouping: 1
cost vector:
(:complex-adds 1)
cost interval: [0 18]
index grouping: 2
cost vector:
(:complex-multiplies 1)

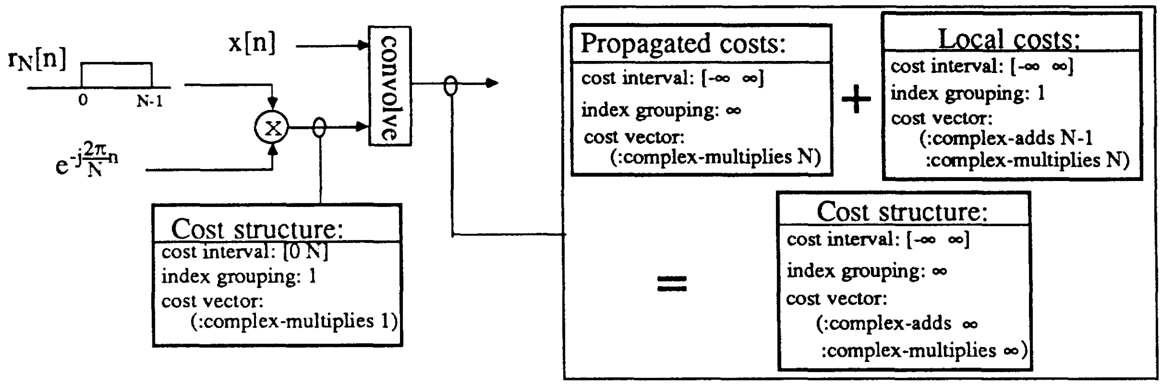
+

Cost structure:
cost interval: [-6 6]
index grouping: 3
cost vector: (:real-adds 1)
cost interval: [6 18]
index grouping: 4
cost vector:
(:real-multiplies 1)

=

Cost structure:
cost interval: [-6 0]
index grouping: 3
cost vector:
(:complex-adds 3
:real-adds 1)
cost interval: [0 6]
index grouping: 6
cost vector:
(:complex-multiplies 3
:real-adds 2)
cost interval: [6 18]
index grouping: 4
cost vector:
(:complex-multiplies 2
:real-multiplies 1)

a) Since the computational costs of the system inputs are propagated to the system output, cost structures will need to be added together. With this need arises the possibility of having to add cost intervals of unequal block sizes. One possible approach to this task, shown here, is to use the least-common-multiple block size as the output block size and add the appropriately scaled cost vectors.



Propagated costs:
cost interval: [-∞ ∞]
index grouping: ∞
cost vector:
(:complex-multiplies N)

+

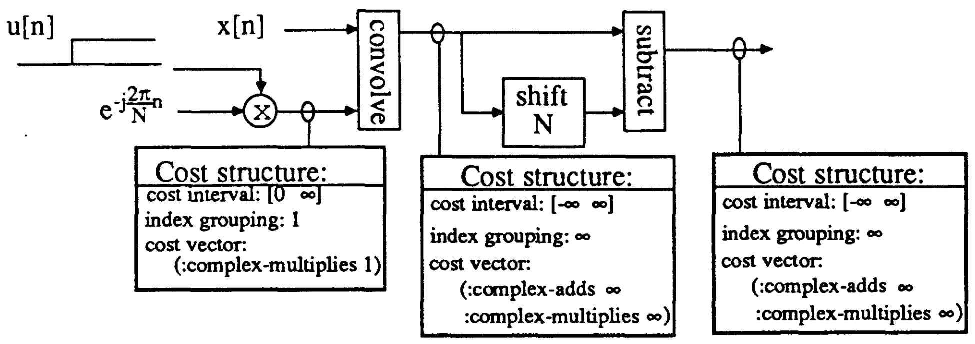
Local costs:
cost interval: [-∞ ∞]
index grouping: 1
cost vector:
(:complex-adds N-1
:complex-multiplies N)

=

Cost structure:
cost interval: [-∞ ∞]
index grouping: ∞
cost vector:
(:complex-adds ∞
:complex-multiplies ∞)

Cost structure:
cost interval: [0 N]
index grouping: 1
cost vector:
(:complex-multiplies 1)

Using least-common-multiple block sizes to add corresponding cost intervals the discrimination of the cost structures is greatly reduced when one of the block sizes is infinite. For example, using this approach to the addition, there is no discrimination between the costs of the implementation in part (b), shown above, and that of the implementation in part (c), shown below.



Cost structure:
cost interval: [0 ∞]
index grouping: 1
cost vector:
(:complex-multiplies 1)

Cost structure:
cost interval: [-∞ ∞]
index grouping: ∞
cost vector:
(:complex-adds ∞
:complex-multiplies ∞)

Cost structure:
cost interval: [-∞ ∞]
index grouping: ∞
cost vector:
(:complex-adds ∞
:complex-multiplies ∞)

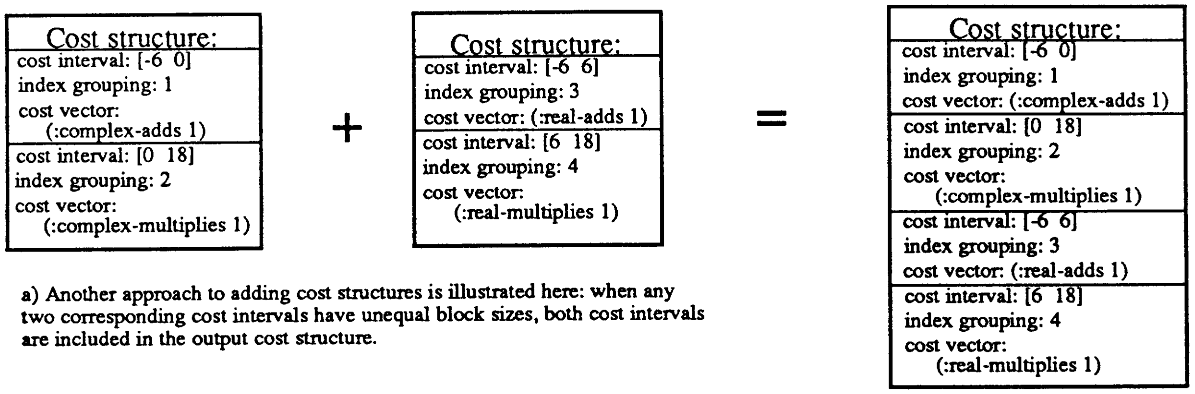
Figure 7-8: Addition of two cost structures using the least-common-multiple block size to add corresponding cost intervals

to reflect computational efficiency can be greatly reduced.

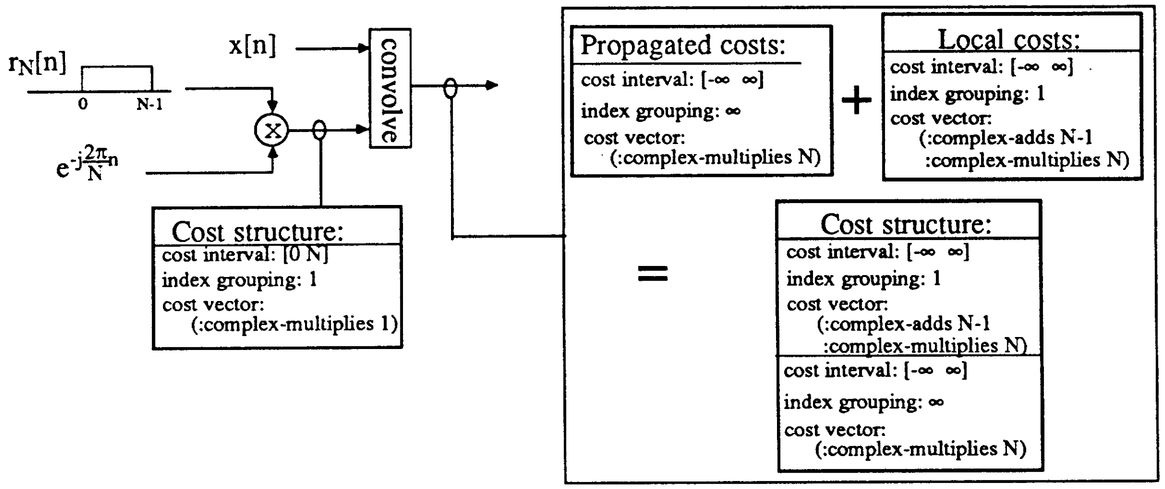
A more appropriate approach to adding cost structures which have unequal corresponding block sizes is maintain separate cost intervals in the output cost structure for each of the block sizes (Figure 7-9-a). Returning to the example used above, the cost structures for the two alternate implementations would each have two cost intervals. The implementation $y[n] = x[n] * (e^{-j\frac{2\pi}{N}kn}r_N[n])$ has a cost structure with N multiplications and $N-1$ additions per sample and with N multiplications per ∞ samples, both covering the full indexing domain (Figure 7-9-b). The implementation $y[n] = x[n] * h[n] - x[n-N] * h[n-N]$ would have a cost structure with ∞ multiplications and ∞ additions per sample and with ∞ multiplications per ∞ samples, both covering the full indexing domain (Figure 7-9-c). Unfortunately, this approach further complicates the determination of dominance between cost structures. The final approach that is actually used in ADE in determining dominance is described in Appendix D.

7.2.2 Local assignment of cost

From the previous subsection, the computation of the output signal incurs both the local cost of the system and the cost of associated with computing the required samples of the input signal. The previous subsection assumed that, after appropriate modification of the input cost structures, the additional cost incurred by the input signal computations was incorporated directly into the cost structure of the output signal by simply adding cost structures. The difficulty with this approach is that it results in artificially high estimates of the computational cost for some implementations. For a simple example, consider the computational cost of a 4-point short-time Fourier transform (Figure 7-10-a). If the cost of computing the inputs to each of the systems is incorporated directly into the cost of the output of the system, the costs of the first stage computations would be incorporated into the STFT cost twice and the cost of computing the input sample values would be counted sixteen times. This type of error can easily affect the dominance relations between implementations: an implementation which is actually computationally



a) Another approach to adding cost structures is illustrated here: when any two corresponding cost intervals have unequal block sizes, both cost intervals are included in the output cost structure.



This approach to adding cost intervals with unequal block sizes allows for discrimination between the costs of the implementation in part (b), shown above, and that of the implementation in part (c), shown below.

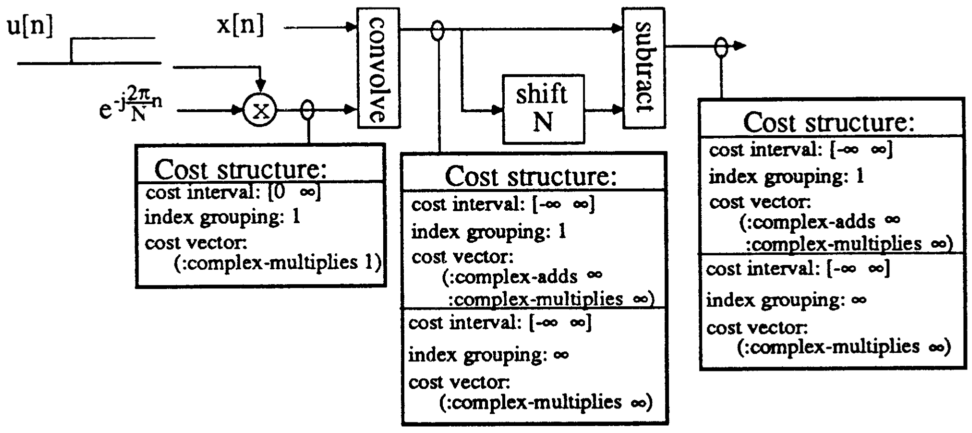
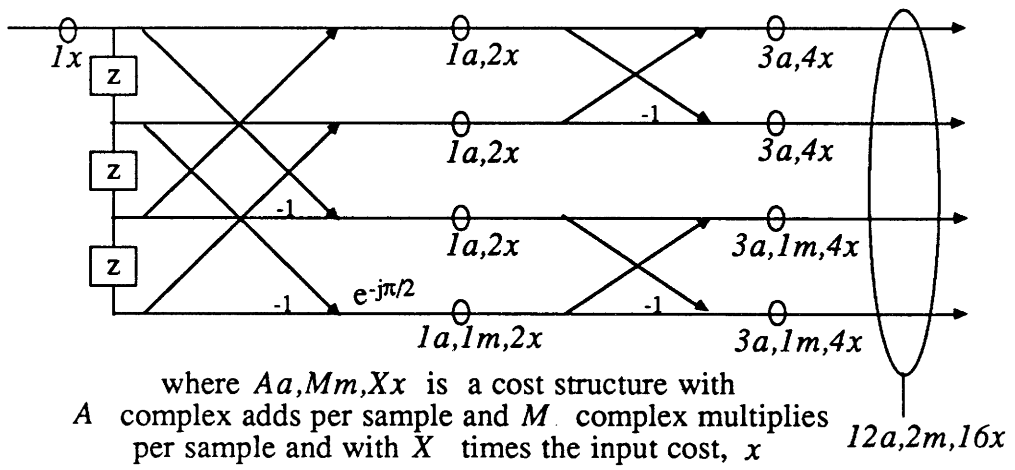
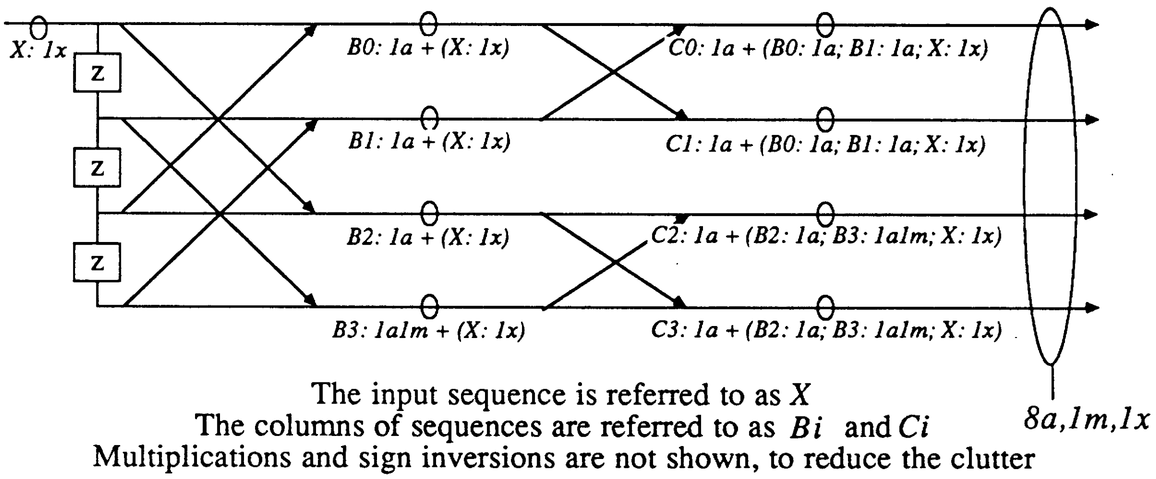


Figure 7-9: Addition of two cost structures using explicit inclusion to add corresponding cost intervals of unequal block sizes



1x
z
z
z
1a,2x
1a,2x
1a,2x
1a,1m,2x
-1
-1
-1
-1
$e^{-j\pi/2}$
3a,4x
3a,4x
3a,1m,4x
3a,1m,4x
where Aa,Mm,Xx is a cost structure with A complex adds per sample and M complex multiplies per sample and with X times the input cost, x
12a,2m,16x

a. If the computational requirements imposed by the inputs to a system are incorporated directly into the cost structure of the output from the system, the cost of signals which are used more than once by a system will be included multiple times. This effect is shown here for a 4-point short-time Fourier transform.



X: 1x
z
z
z
B0: 1a + (X: 1x)
B1: 1a + (X: 1x)
B2: 1a + (X: 1x)
B3: 1a1m + (X: 1x)
C0: 1a + (B0: 1a; B1: 1a; X: 1x)
C1: 1a + (B0: 1a; B1: 1a; X: 1x)
C2: 1a + (B2: 1a; B3: 1a1m; X: 1x)
C3: 1a + (B2: 1a; B3: 1a1m; X: 1x)
8a,1m,1x
The input sequence is referred to as X
The columns of sequences are referred to as Bi and Ci
Multiplications and sign inversions are not shown, to reduce the clutter

b) By maintaining separate cost structures for each of the inputs plus a cost structure for computations local to the system itself, inputs which are encountered multiple times need not have their cost structure included more than once in the overall cost. By this mechanism, the classic order N log N cost for the FFT can be determined.

Figure 7-10: Multiply-used signals as a motivation for consignment of costs to the signals or systems that incur them

efficient can easily be overshadowed by another implementation due to these errors.

To avoid this problem, computational costs are internally associated with the signal or system which locally incurred them. Then, simply by cross-checking this information, the costs of signals which are used more than once need not be included in the overall cost multiple times. Only when the overall costs of the system output is required, either to determine dominance or to respond to the user, are the cost structures summed. An example of this is shown in Figure 7-10-b, for the same 4-point short-time Fourier transform examined above.

Fortunately, since the cost structures of inputs and local costs are added together prior to determination of dominance, this local assignment of costs does not affect the procedure used to determine dominance. In this way, maintaining a local assignment of costs provides additional accuracy in the combination of the cost structures without increasing the complexity of their external behavior or their ranking.

7.3 Summary

This chapter has developed a framework for computing and comparing cost measures for system implementations. Starting with a simple vector of operation counts and memory requirements, modifications have been made to increase the accuracy of the cost measures. In summary, each cost measure is a distribution over the indexing variable of these simple cost vectors. In particular, a "cost interval" associates a cost vector with an indexing interval and an indexing block size. The cost of computing a block of samples within the indexing interval is given by the cost vector. Moreover, the cost of computing one sample within the indexing interval is the same as the cost of computing the full block of samples within the indexing interval: that is, the computation of the samples is "blocked" together. Finally, there can be more than one cost interval covering a single indexing interval, due to the possibility of differing block sizes within the interval. In this case, the cost of computing a set of samples within the interval is the sum of the costs implied by the covering cost intervals. Internally, costs are propagated from the system inputs to the system output by modifying the input cost structures according to the input/output dependencies. After modifying the input cost structures to reflect the viewpoint of the system output, these modified cost structures are associated with the signal that locally incurred them. Only when system dominance is being determined or user queries are being answered are the cost structures combined. The additional information provided by this local assignment of costs allows the costs of systems which use a single input multiple times to be correctly determined. The addition of cost structures is completed by adding corresponding cost intervals that have equal block sizes and by explicitly including all of the corresponding cost intervals that have unequal block sizes.

Chapter 8

Contributions and Limitations

As stated at the beginning of this thesis, the long-term goal of research in the area of signal-processing algorithm manipulation is to eventually provide the engineer with a single, integrated computer environment which supports and expedites all stages of the design process. This thesis has addressed some of the issues associated with providing such a computer environment for signal-processing algorithm design. The foundations used in this research were built in Kopec (1980), Dove et al. (1984) and Myers (1986).

Kopec (1980) reported one of the first efforts at providing a set of explicit representations for signal processing. This explicit representation of sequences as distinct objects is one of the major contributions of Kopec (1980). Two other equally important contributions of Kopec (1980) are sequence immutability and the uniform external interface to the sequences. Mathematically, sequences have an explicit identity which is immutable: the identity and the sample values of any given sequence are completely fixed. In addition, mathematically, any of the sample values of a sequence can be referenced independent of the algorithm provided for computing the sample values. These were all characteristics which Kopec (1980) advocated in signal representations. The research in Dove et al. (1984) and Myers (1986) refined and extended this numeric representation of specific discrete-time sequences.

Many of the contributions of Myers (1986) lie in the area of symbolic signal representation. Myers (1986) introduced representations for discrete-time Fourier-transform signals, allowing expression manipulations to occur both in the time and the frequency domains. Myers (1986) also introduced representations for incomplete signal descriptions: abstract signals could then be manipulated, providing the ability to manipulate whole classes of signals simultaneously. Most importantly, using these representations, Myers (1986) demonstrated the ability of the computer to symbolically manipulate signal-processing expressions: an algorithm was autonomously uncovered which was subsequently, independently presented as an efficient implementation for non-integer sampling rate conversion.

8.1 Contributions

This thesis has introduced a variety of refinements and extensions to the work in Myers (1986). However, the major contribution of this research lies in its efforts to limit the combinatorial growth of the search for efficient implementations. Two other important contributions of this work are the development of accurate cost measures including an accurate way of propagating and combining costs and the development of signal and system representations which allow information to be easily and efficiently shared between related objects.

The internal regularity of signal-processing algorithms was used to limit the size of the search space for equivalent implementations. This regularity in the low-level signal-processing descriptions is commonly pointed out using information provided by the high-level description of the same operation. Without these constraints, many FFT-based and polyphase-based algorithms would be beyond the scope of consideration, due to the combinatorial expansion of these design spaces.

The time distribution of costs are described using "cost intervals": these descriptions include information about the extent over which a cost applies; the blocking of samples within that extent; and the vector of operation counts and memory requirements for each block. The possibility of shared subexpressions allows the cost description of separate parts of a single algorithm to interact. These interactions are reflected in the final addition of the component costs to determine the total cost of an implementation. Finally, bounding constraints for comparing the relative costs of distinct operations, such as complex and real multiplications, were added to increase the area of dominance of each implementation.

The two-level representation of abstract objects allows information about property values to be shared between related abstract instances. A similar two-level representation was developed for symbolically constrained objects which depend on a single abstract object. The advantage of these two-level representations is the ability to reuse information derived for one instance in characterizing a related instance.

In addition to these primary contributions, this research effort also provides some other contributions which are perhaps of lesser importance.

This thesis has developed a signal-processing environment which explicitly separates the description of the control structure, the general description of the properties and the description of the signals and systems.

The numeric classifications and representations of signals and systems were integrated with their symbolic classifications and representations. A common class hierarchy for both types of representations was developed.

The interfaces for properties and sample values were made uniform across all classifications of signals and systems: simple specific, symbolically constrained, and abstract; discrete-time domain, Fourier domain and z domain; and computable and uncomputable. Symbolic numbers are used to represent the sample values of abstract signals, symbolically constrained signals and uncomputable signals.

The class hierarchy of signals and systems was further exploited to provide a hierarchical structure to the rule base and to potentially reduce the number of typing tests performed within each rule. This hierarchical organization reduces the apparent size of the rule base: the only rules which are considered in a search are those which apply to the same class as or a superclass of the current object.

8.2 Limitations

The FSK-code detector has been used in this thesis to illustrate the power both of symbolic manipulation and of parallel constraints. In this section, a final design example is considered primarily to illustrate some of the limitations of the described environment. Additional limitations are pointed out in the next section on areas for future research.

This example considers the recovery of the in-phase and quadrature samples of an RF signal, such as was used as the front end of the FSK-code detector shown in Figure 3-1. The conventional structure for the recovery of in-phase and quadrature samples from an RF signal is shown in Figure 8-1. Two base-band analog channels are created by separately mixing the input channel with the in-phase and quadrature components of the carrier frequency and then low-pass filtering. These base-band signals are sampled at their Nyquist rate B to generate the I and Q samples. This approach requires matching between the frequencies and phases of the two mixers; between the gains on the two analog low-pass filters; and between the gains and the rates of the two A/D converters.

An alternative approach to this recovery was suggested by Rader (1984): the proposed approach avoids the problem of matching analog components. As shown in Figure 8-2, the use of two analog channels is avoided by not immediately beating the RF signal down to base band. Instead, the signal is beat down to a center frequency of B where B is the bandwidth of the RF band and this signal is sampled at a rate of $4B$. The resulting discrete-time sequence is passed through a 90°-phase splitter network, designed to pass the positive frequency band and to remove the negative frequency band: thus, using $i_{\uparrow 4}[n]$ and $q_{\uparrow 4}[n]$ to represent the two outputs from the phase-splitter network, the sequence $i_{\uparrow 4}[n] + j\ q_{\uparrow 4}[n]$ will ideally only have energy in the frequency interval from $\frac{\pi}{4}$ to $\frac{3\pi}{4}$. Since this remaining energy lies within an interval which is only $\frac{\pi}{2}$ wide, the sequence can be downsampled by four, resulting in the output sequences, $i[n] = i_{\uparrow 4}[4n]$ and $q[n] = q_{\uparrow 4}[4n]$, at the same rate as the output sequences from the conventional structure. When the digital portion of the structure shown in Figure 8-2 was given to ADE to find an efficient implementation, the structure shown in Figure 8-3 was found as

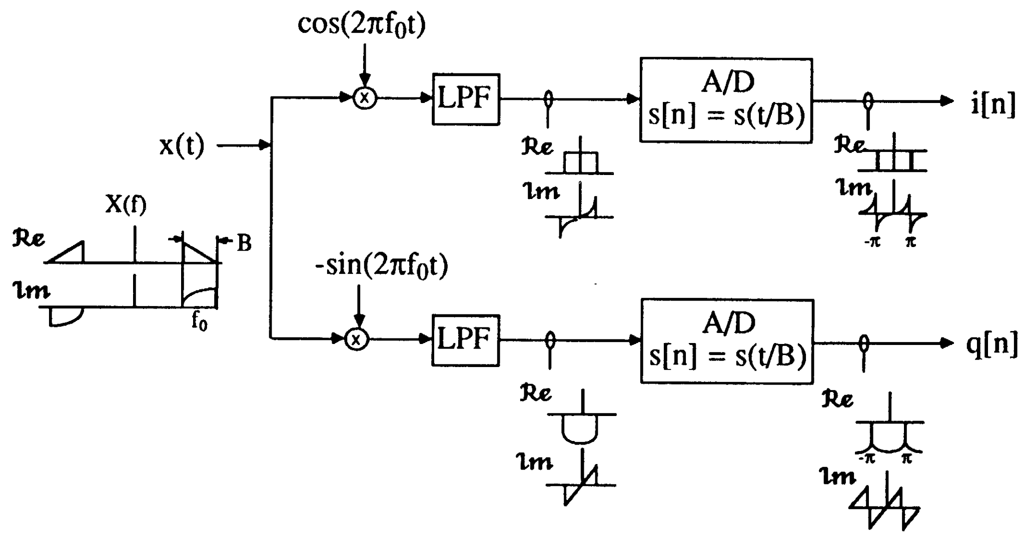
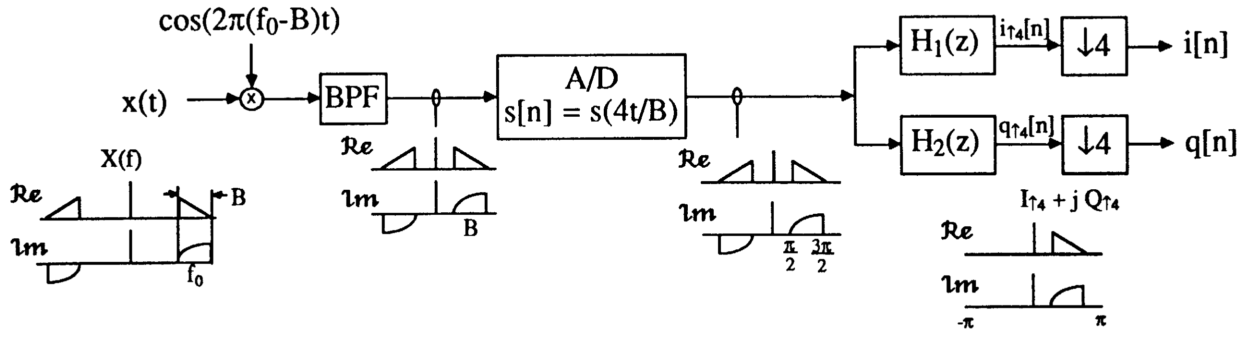


Figure 8-1: The conventional approach to recovery of I and Q samples of an RF signnal

The conventional RF sampling structure is shown here along with pictorial Fourier-domain representations of the intermediate signals



$$H_1(z) = z^{-1}\left(\frac{z^{-2}-a^2}{1-a^2z^{-2}}\right) \qquad H_2(z) = \left(\frac{z^{-2}-b^2}{1-b^2z^{-2}}\right)$$

Figure 8-2: An alternate approach to recovery of I and Q samples of an RF signal

This alternative for RF sampling was proposed by Rader (1984). Pictorial Fourier-domain representations of the intermediate signals are included.

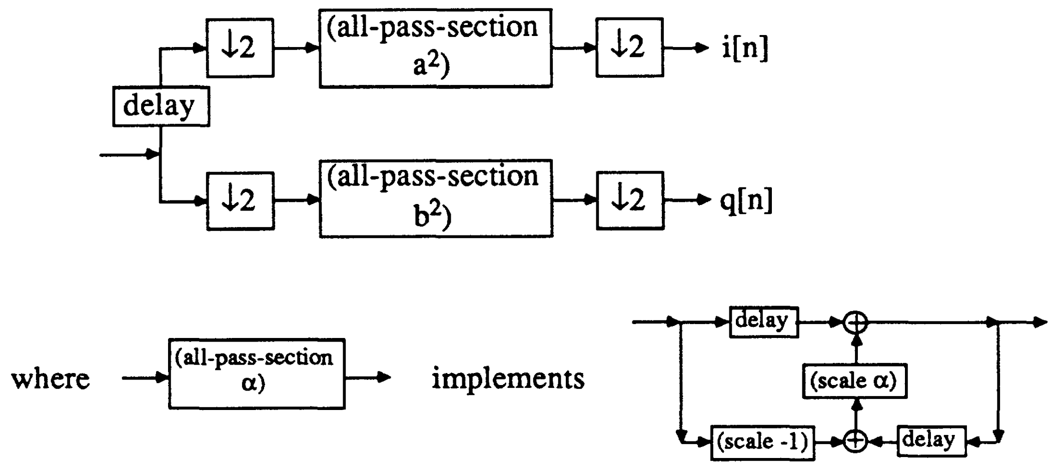


Figure 8-3: An efficient implementation of the digital portion of the structure shown in Figure 8-2

the most efficient implementation for this structure. This efficient implementation can also be found in Rader (1984).

Part of the design process completed by Rader included the recognition of the difficulties inherent in the conventional structure shown in Figure 8-1: before the search for a new implementation began, the limitations of the current implementation had to be recognized. These limitations do not occur in terms of computational costs, which have been used throughout this thesis for comparing alternate implementations. Instead, the difficulty with the conventional implementation lies in the noise introduced by mismatches between the two analog channels. To model this source of noise, not only does the noise inherent in individual mixers, analog filters and analog-to-digital converters need to be modeled, but the signal distortion introduced by their mismatch must be analyzed. This analysis requires models of the "reproducibility" of the parameter values of the operators: for example, a model would be needed for the difficulty in matching the frequencies of the two mixers and in matching their initial phases so that they run in quadrature.

This points out two areas which could use further development in ADE. The first area is in the representation of continuous-time operations: ADE does not include any representation of analog operations. Although discrete-time Fourier-transform signals are represented in ADE, distinctions should be made between these continuously-indexed signals and continuous-time signals. The relationships between continuous-time signals and discrete-time sequences is fundamentally different than between discrete-time Fourier-transform signals and discrete-time sequences: in particular, the first relationship is a many-to-one mapping through A/D sampling and the second relationship is a one-to-one function through the inverse Fourier transform.

The second area for development which this points out is in modeling the noise within a system. The noise within the computation should be modeled to allow general analyses of the sensitivity of different implementations. However, it is the reproducibility of the system parameters which must be modeled for this particular problem. Both of these modeling operations are difficult, since the type and the level of noise with the system is often closely tied to the details of the hardware implementation.

The most impressive part of the design completed by Rader was his ability to consider the problem as a whole, looking both at the original difficulties which he needed to avoid and at the interactions between the choices of intermediate signals, sampling rates and filter structures. The original difficulties in matching the analog channels constrained his choices of intermediate analog signals: the signal needed to be real, since only one analog channel was desired and since only real-valued analog signals and operations could be used. Given this constraint, the intermediate analog signal could be given any carrier frequency above $\frac{B}{2}$ and this analog signal could be sampled at any rate above its Nyquist rate. Rader's choice of the center frequency and the sampling rate allowed him to use a low-order, all-pass filter network to implement a phase splitter. This insight results both from the global consideration of the problem at hand and from extensive knowledge of the tools available.[1]

[1]The transformation from the conventional implementation to the structure proposed by Rader (1984) also requires the use of an approximation. In particular, due to the phase characteristics of the digital

Obviously, this type of insight is difficult to include in any environment. This type of insight requires both extensive knowledge of the types of operations which can be efficiently implemented and the ability to select from this store of knowledge without attempting an exhaustive search. A large store of knowledge has been included in ADE and more can always be added. Instead, the basic difficulty lies in encoding this knowledge in a way that allows the environment to recognize opportunities like this and in creating an environment which will modify the parameter selection on the various subproblems to exploit these opportunities.

8.3 Suggestions for Future Research

Returning to the stated long-term goal of this research, a number of issues must be addressed before an integrated signal-processing design environment supporting the full design process can be realized. This section attempts to outline the areas in which further research is required to achieve this goal.

8.3.1 Signal and system representation

The representations for multi-dimensional signals and systems is a largely unexplored area of research. Some of the issues in representing multi-dimensional signals include the description of their non-zero support, the description of their symmetry characteristics and the description of their cost. Representing the non-zero support and the cost of a multi-dimensional signal requires the description of a multi-dimensional region: rectangular regions are often useful in describing image data while annular regions are used by z-transform signals. In addition, irregularly shaped regions of support would be useful in describing the segmentation of an image or the Nyquist volume of a high-definition television signal.

filtering network, the output from the structure shown in Figure 8-2 has an added nominal time delay of $\frac{7}{16}$ samples and a frequency-dependent deviation from this nominal time delay of $\frac{1}{16}$ samples. Although no approximate transformations are currently included in ADE, their inclusion does not present any immediate conceptual difficulties.

The representation of explicitly recursive definitions is another unexplored area of research. To support the recursive definition of signals, provisions would need to be made for deferring the evaluation of the system inputs: for example, in a simple feedback loop, a representation of the output signal must be provided *before* the addends which close the loop can be evaluated. The representation of recursive operations would be simplified if the index parameter of the signal were explicitly represented. Furthermore, the derivation of recursive forms is not possible without the explicit representation of the index parameter. To illustrate, consider the design of the bank of matched filters for the rectangularly windowed frequency chips. As was pointed out in section 3.4 of Chapter 3, an alternate implementation of this filter bank uses a recursive formulation of each of the matched filter computations. In particular, the computation

$$X[k,n] = \sum_{m=0}^{N-1} x[n+m]e^{-j\frac{2\pi}{N}km}$$

can be completed using the recursive formulation

$$X[k,n] = e^{-j\frac{2\pi}{N}k}(X[k,n-1] + x[n+N-1] - x[n-1])$$

The derivation of this recursive formulation requires the manipulation of the index values involved in the sample-value description. This type of manipulation is not supported by the current signal representations, since there is no explicit representation of the index parameter: informally, the current representation supports consideration of x but not of $x[n]$.

8.3.2 Noise and sensitivity analyses

The analysis of the sensitivity characteristics of an algorithm is not supported in either E-SPLICE (Myers, 1986) or ADE. The noise within the computation of system should be modeled to allow general analyses of the sensitivity of different implementations. In addition, as pointed out in the previous section, the reproducibility of the system parameters should also be modeled. Both of these modeling operations are difficult, since the type and level of noise with the system is often closely tied to the details of hardware implementation.

8.3.3 Cost measures

Another area of research lies in improving the cost measures still further. As shown by the example in Figure 7-7, some comparatively simple cost dependencies can not be adequately described using the current representation. One alternate approach to cost propagation is suggested by analogy with the work by Zissman (1986). Zissman (1986) describes software tools which convert block diagrams into assembly code for a MIMD array. To distinguish between synchronous and asynchronous operations, Zissman (1986) includes explicit operations for blocking and unblocking his streams of data. A similar approach could be taken to the propagation of costs. In particular, instead of altering the input cost structures according to the propagation characteristics of the system, the input cost structure could be carried forward unchanged along with a description of the propagation characteristics of the intervening systems. Thus, the blocking imposed by the system would be an explicit part of the propagated cost description. This would allow overlapping cost dependencies, such as those shown in Figure 7-7-c, to be described without any undue "cost inflation."

Both E-SPLICE (Myers, 1986) and ADE have avoided all consideration of the selection and scheduling onto a processor architecture. While Prasanna (1988) and Fogg (1988) both propose to investigate this area, a significant amount of effort will still need to be devoted to hardware selection and utilization. The most immediate area which must be addressed to integrate this selection process with the algorithm selection is the improved characterization of the cost of an implementation. The current cost metric implicitly assumes serial implementations of the algorithms: comparisons of operation counts and memory references are only valid when the evaluation is completely sequential. A cost metric should be developed for parallel architectures, giving some measure of the internal regularity of the algorithm and the computational requirements along what is likely to be the critical path.

8.3.4 Regularity constraints

As pointed out in Chapter 6, often there are a large number of computationally efficient implementations which will not be found when regularity constraints are enforced: if the equivalent subexpressions of two computationally efficient implementations can be intermixed without reducing the computational efficiency, then the resulting structure will be another computationally efficient implementation. One possible way of obtaining these structures while still using regularity constraints to reduce the size of the search space is to enforce the regularity constraints throughout the search for efficient implementations and to then interchange subexpressions between the discovered implementations. In particular, when two or more efficient implementations involve equivalent subexpressions, the subexpression seen in one implementation can be used to replace the equivalent subexpression in the other implementation, without regard to any previously enforced regularity constraint. This will have the effect of giving the outer product of the implementations of the subexpressions but will not cause a corresponding expansion of the search space, since the search process will have already terminated. This effect of considering the outer product of implementations without going through a full search is highly reminiscent of the decoupled design strategy proposed by Fogg (1988).

Another area which could certainly be developed further is the automatic detection of regularity constraints. In ADE, the regularity constraints which can be placed on a signal-processing algorithm must be explicitly pointed out: propagation of these constraints, both within the algorithm and to modified expressions, is supported by the environment but the initial description of the constraints must be done manually. One of the drawbacks of this approach was illustrated with the modulated Hanning-window filter bank: the author was forced to intervene in the search for constrained equivalent forms, to point out a new correspondence constraint. If the environment searched for and uncovered these internal correspondences itself, this artificial intervention would not be necessary.

Another area associated with regularity constraints which could be developed further is the manipulation of nearly regular expressions. If a signal-processing expression is defined without consideration of internal regularity, the expression will often be nearly but not completely regular. An example of this "near regularity" is provided by the conventional short-time Fourier-transform structure. The short-time Fourier-transform structure shown in Figure 6-6 includes scaling operations which would normally be omitted: in particular, in the conventional FFT structure, scaling operations are only included on half of the butterfly outputs. To provide a completely regular structure, Figure 6-6 inserts identity scaling operations to balance the twiddles. The automatic detection of this type of near regularity and the automatic insertion of the appropriate identity operations is another area is another potential area for research.

8.3.5 Automatic extension of the rule-base

Another area which should be considered is the automatic extension of the rule-base. In the normal course of processing, the design environment will reach a multitude of conclusions about the signals and systems which it has been asked to manipulate: for example, the simplifications, non-zero supports and sample types of various signals will be determined. These conclusions will be arrived at in the course of answering the user's queries: this is their primary motivation. These conclusions can be further exploited to increase the efficiency of subsequent processing. In particular, if the sequence of rules used in arriving at a conclusion is recorded, this information can be exploited.

The backtrace could be used in generating a new rule which is the composition of the traced rule sequence. The basic process of generating new transformation rules by generalizing a successfully applied series of transformations was first explored by Fikes and Nilsson in STRIPS (Fikes and Nilsson, 1971). STRIPS is a problem-solving program for a robot operating in a world of rooms, doors and boxes. A later edition of the basic STRIPS system permitted plans to be generalized and reused. By analogy with STRIPS, the desired approach is to generalize the numbers, signals and systems included in the backtrace of rules. The advantage of generating these composite rules is the possible savings in both time and memory: time may be saved since multiple steps would be completed simultaneously and memory may be saved since intermediate signals and systems will not be generated.

In addition, if an identity loop is uncovered in the course of generating equivalent forms, this new identity transformation can be used to generate a new simplification rule. The possibility of identity loops in searches for equivalent forms was first discussed in Chapter 6. An identity loop occurs when the search space for the equivalent forms of a subexpression coincides with the search space for the original signal-processing expression. Given this coincidence, the composite transformation for traveling from the subexpression to the original expression is an identity transformation. The strategy which detects and breaks these identity loops can reconstruct this identity transformation and this information can then be formulated as a simplification rule.

A preliminary exploration of these two possibilities has been completed. However the potential of this part of the research remains to be demonstrated. One difficulty is that, for the rules generated from the backtrace of a previous conclusion, the information which is being encoded in the composite rule is actually already available in the rules given in the backtrace. The use of identity loops has the potential for uncovering new information: a completely new simplification may be generated, since the information which is being used here is actually the composite of equivalent-form transformations as opposed to simplification transformations. This possibility has yet to be demonstrated.

A slightly different approach to the generation of simplification rules from backtraces could be fruitful. Some signal-processing descriptions which, to the engineer, are obviously simpler than the original description can be uncovered using a series of equivalent-form transformation rules: Figure 8-4 shows one such example. If some reasonable measure were developed for what represents a simplification, the generation of a simplification rule from this series of equivalent-form transformations would be possible. An example of such a measure would be the number and identity of the input signals and the compo-

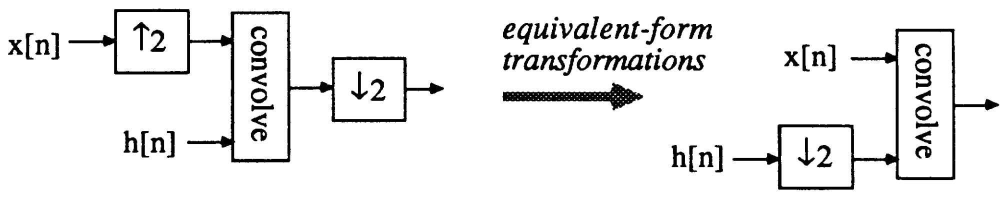


Figure 8-4: The results from a series of equivalent-form transformations which could be used as a simplification

A slightly different approach to the generation of simplification rules from backtraces may also be fruitful. Some signal-processing descriptions, like the one shown here, are to the signal processor obviously simpler than the original description. These "simpler" descriptions can often be obtained using a series of transformation rules. If some reasonable measure were developed for what represents a simplification, the generation of a simplification rule from this series of transformations would be possible.

nent systems used within the signal-processing expression. If the input signals and the component systems used in the result of a series of equivalent-form transformations is a strict subset of those used in the starting object, then the transformation could be made into a simplification. The example in Figure 8-4 meets this criteria: all the input signals and the component systems which are used in the final signal-processing expression are present in equal or greater numbers in the original expression and there is at least one input signal or component system used in the original expression which either is not present or is present in fewer numbers in the final expression. Unfortunately, this measure of simplicity needs modification, since it would simplify all polyphase implementations of downsampled convolutions to the direct implementation of the downsampled convolution. To avoid this type of error, the measure of simplicity should take the computational requirements of the alternate implementations into account as well as the number and identities of the input signals and the component systems.

Appendix A

The Algorithm Design Environment (ADE): a user's guide

ADE is a descendant of the SPLICE and E-SPLICE environments. ADE inherits its basic approach to signal definition and representation from SPLICE (Dove et al., 1984; Myers, 1986). The influences of E-SPLICE (Myers, 1986) and to a lesser extent PDA (Dove, 1986) are reflected in parts of the rule base. In particular, E-SPLICE used backward-chaining rules in describing some of the properties of signals and supported subpattern matching within the patterns of these rules. The approach which is used in ADE to testing forward-chaining rules with multiple matching patterns which is introduced by Dove (1986).

ADE also makes use of a subset of QM (Sacks, 1982) and of a limited number of functions from MACSYMA (Mathlab Group, 1983). QM (Sacks, 1982) is the product of research into qualitative mathematics. It represents, manipulates and describes piecewise-continuous functions. A subset of QM is used to record and propagate constraints on symbolic numbers. ADE includes an extension to QM to support limited reasoning about symbolic integers as well as the continuously variable numbers. ADE also makes limited use of MACSYMA (Mathlab Group, 1983) to simplify and factor polynomials.

ADE is written in Symbolics Common Lisp (Symbolics, 1986). This choice of language provides both the flexibility of a LISP dialect and support for object-oriented programming. In its purest forms, LISP is distinguished from other languages in its uniform representation of data and functions. The basic data structure in LISP is the list, an ordered collection of elements. By representing functions using lists, LISP facilitates the manipulation of functional forms as data.

To provide some measure of size, ADE contains 26 properties, for characterizing signals and systems; 129 classes of systems, each with an associated output signal class; and an additional 43 classes of inherent signals. The signal and system definitions alone are encoded in ten files containing about 7800 lines of code. The remaining sixty files contain functions for defining classes; functions describing the control structure of the environment; and functions for the manipulation of symbolic numbers, intervals, polynomials, symmetry descriptors and costs.

Within this appendix, the functions currently available in ADE for the external description and manipulation of signals and systems are described first. These functions include the functions for creating and retrieving signals and systems; the functions for retrieving signal and system properties; the functions for retrieving sample values from signals; and the functions for creating, and manipulating intervals, symbolic numbers and polynomials. The functions available in ADE for extending or modifying the environment are described in the second half of this appendix. These functions include functions for describing new properties; functions for describing new signal or system classes; and functions for describing new control strategies.

The format used in describing these functions is:

FUNCTION-NAME *input-list*

with *[ ]* indicating one or more optional arguments.

The format used to describe a system and its output signal is:

(SYSTEM-CLASS-NAME *parameter*$_1$ ... *parameter*$_N$) *input-list* where evaluating (SYSTEM-CLASS-NAME *parameter*$_1$ ... *parameter*$_N$) would return the system and where requesting the OUTPUT-OF the system applied to the *input-list* would return the output signal from the system.

A.1 Functions for Creating and Manipulating Signals and Systems

Specific signals and systems can be created and retrieved using:

SPECIFIC-MEMBER *(class [parameter$_1$ ... parameter$_N$])*
The system or inherent signal generated by applying *class* to *parameter*$_1$ ... *parameter*$_N$. *class* must one of the system or inherent signal classes which can generate specific systems or signals. These classes are listed in Tables A.1.1 and A.1.2.

OUTPUT-OF *(system [input$_1$ ... input$_N$])*
The output signal generated by applying *system* to *input*$_1$ ... *input*$_N$.

Abstract signals and systems can be created and retrieved using:

A-MEMBER-OF *(class [&properties property$_1$ value$_1$ ... property$_N$ value$_N$])*
An abstract instance of *class* with *property*$_1$ being *value*$_1$, ..., *property*$_N$ being *value*$_N$. *class* must one of the system or inherent signal classes which can generate abstract systems or signals. These classes are listed in Tables A.3 and A.4.

A.1.1 Specific, inherent signal classes

The inherent signal classes currently provided in ADE which can generate specific signals are listed in Table A.1. These classes can not be used to generate abstract signals. Instead, the generalization must occur in their parameter values.

The specific, inherent signals in these classes can be created using SPECIFIC-MEMBER as described above. These specific, inherent signals can also be created and retrieved using the following functions.

Table A.1: Signal classes in ADE containing specific, inherent signals

RATIONAL-ZT	CAUSAL-RECTANGULAR-WINDOW-SEQUENCE
CONSTANT-SEQUENCE	CAUSAL-RECTANGULAR-WINDOW-SIGNAL
CONSTANT-SIGNAL	CAUSAL-RECTANGULAR-WINDOW-2D
CONSTANT-ZT	RECTANGULAR-WINDOW-SEQUENCE
CONSTANT-2D	RECTANGULAR-WINDOW-SIGNAL
2D-CONSTANT-1ST-D	RECTANGULAR-WINDOW-2D
2D-CONSTANT-2ND-D	SINC-SEQUENCE
POWER-SEQUENCE	SINC-SIGNAL
IMPULSE-SEQUENCE	COSINE-SEQUENCE
IMPULSE-2D	SINE-SEQUENCE
GENERAL-EXPONENTIAL-SEQUENCE	CAUSAL-HAMMING-WINDOW-SEQUENCE
GENERAL-EXPONENTIAL-SIGNAL	FIR-SEQUENCE
COMPLEX-EXPONENTIAL-SEQUENCE	IIR-SEQUENCE
COMPLEX-EXPONENTIAL-SIGNAL	CAUSAL-IIR-SEQUENCE
UNIT-STEP-SEQUENCE	ANTICAUSAL-IIR-SEQUENCE
CAUSAL-HANNING-WINDOW-SEQUENCE	STABLE-IIR-SEQUENCE

RATIONAL-ZT *(numerator denominator roc)*
A rational z-transform signal, $X(z) = \frac{N(z)}{D(z)}$, where *numerator* gives the polynomial $N(z)$ and *denominator* gives the polynomial $D(z)$, with convergence on $|z|$ in the interval *roc*. *numerator* and *denominator* must be polynomials and *roc* must be an interval starting at or above zero.

CONSTANT-SEQUENCE *(value)*
A constant sequence, $x[n] = v$, where *value* gives v. *value* must be a number.

CONSTANT-SIGNAL *(value)*
A constant discrete-time Fourier-transform signal, $X(e^{j\omega}) = v$, where *value* gives v. *value* must be a number.

CONSTANT-ZT *(value)*
A constant z-transform signal, $X(z) = v$, where *value* gives v. *value* must be a number.

CONSTANT-2D *(value)*
A constant two-dimensional sequence, $x[n_1, n_2] = v$, where *value* gives v. *value* must be a number.

2D-CONSTANT-1ST-D *(1d-sequence)*
A two-dimensional sequence which is constant in the first dimension, $x_{2D}[n_1, n_2] = x_{1D}[n_2]$, where *1d-sequence* gives $x_{1D}[n]$. *1d-sequence* must be a discrete-time sequence.

2D-CONSTANT-2ND-D *(1d-sequence)*
A two-dimensional sequence which is constant in the second dimension, $x_{2D}[n_1, n_2] = x_{1D}[n_1]$, where *1d-sequence* gives $x_{1D}[n]$. *1d-sequence* must be a discrete-time sequence.

POWER-SEQUENCE *(order)*
A power sequence, $x[n] = n^k$, where *order* gives k. *order* must be an integer.

IMPULSE-SEQUENCE *()*
The impulse sequence, $x[n] = \begin{cases} 1 & n = 0 \\ 0 & otherwise \end{cases}$

IMPULSE-2D *()*
The two-dimensional impulse sequence, $x[n_1, n_2] = \begin{cases} 1 & n_1 = n_2 = 0 \\ 0 & otherwise \end{cases}$

GENERAL-EXPONENTIAL-SEQUENCE *(base)*
A general exponential sequence, $x[n] = b^n$, where *base* gives b. *base* must be a number.

GENERAL-EXPONENTIAL-SIGNAL *(base)*
A general exponential discrete-time Fourier-transform signal, $X(e^{j\omega}) = b^{\omega}$, where *base* gives b. *base* must be a number.

COMPLEX-EXPONENTIAL-SEQUENCE *(frequency)*
A complex-exponential sequence, $x[n] = e^{j\omega_0 n}$, where *frequency* gives ω_0. *frequency* must be a real-valued number.

COMPLEX-EXPONENTIAL-SIGNAL *(frequency)*
A complex-exponential sequence, $X(e^{j\omega}) = e^{jt_0\omega}$, where *frequency* gives t_0. *frequency* must be a real-valued number.

UNIT-STEP-SEQUENCE *()*
The unit step sequence, $u[n] = \begin{cases} 1 & n \geq 0 \\ 0 & otherwise \end{cases}$

CAUSAL-RECTANGULAR-WINDOW-SEQUENCE *(length)*
A causal, rectangular-window sequence, $r[n] = \begin{cases} 1 & 0 \leq n < L \\ 0 & otherwise \end{cases}$, where *length* gives L. *length* must be an integer.

CAUSAL-RECTANGULAR-WINDOW-SIGNAL *(length)*
A causal, discrete-time Fourier-transform, rectangular-window signal, $R(e^{j\omega}) = \begin{cases} 1 & 0 \leq (\omega \bmod 2\pi) < L \\ 0 & otherwise \end{cases}$, where *length* gives L. *length* must be a real number.

CAUSAL-RECTANGULAR-WINDOW-2D *(1st-d-length [2nd-d-length])*
A causal, two-dimensional, rectangular-window sequence, $r[n_1, n_2] = \begin{cases} 1 & 0 \leq n_1 < L_1 \\ & \& \ 0 \leq n_2 < L_2 \\ 0 & otherwise \end{cases}$, where *1st-d-length* gives L_1 and *2nd-d-length* gives L_2. *1st-d-length* and *2nd-d-length* must be integers. *2nd-d-length* defaults to *1st-d-length.*

RECTANGULAR-WINDOW-SEQUENCE *(length [center-p])*
A rectangular-window sequence, $r[n]$. If the window is centered, $r[n] = \begin{cases} 1 & -\frac{L-1}{2} \leq n \leq \frac{L-1}{2} \\ 0 & otherwise \end{cases}$. If the window is not centered, $r[n] = \begin{cases} 1 & 0 \leq n < L \\ 0 & otherwise \end{cases}$. *length* gives L. *length* must be an integer. If the window is centered, *length* must be an odd integer. *center-p* defaults to centered.

RECTANGULAR-WINDOW-SIGNAL *(length [center-p])*
A discrete-time Fourier-transform, rectangular-window signal, $R(e^{j\omega})$. If the window is centered, $R(e^{j\omega}) = \begin{cases} 1 & 0 \leq (\omega \bmod 2\pi) < \frac{L}{2} \\ 1 & 2\pi - \frac{L}{2} \leq (\omega \bmod 2\pi) < 2\pi \\ 0 & otherwise \end{cases}$. If the window is not centered $R(e^{j\omega} = \begin{cases} 1 & 0 \leq (\omega \bmod 2\pi) < L \\ 0 & otherwise \end{cases}$. *length* gives L. *length* must be a real number. *center-p* defaults to centered.

RECTANGULAR-WINDOW-2D *(1st-d-length [2nd-d-length] [1st-d-center-p] [2nd-d-center-p])*
A two-dimensional, rectangular-window sequence $r[n_1, n_2]$. If the window is centered, $r[n_1, n_2] = \begin{cases} 1 & -\frac{L_1-1}{2} \leq n_1 \leq \frac{L_1-1}{2} \\ & \& \ -\frac{L_2-1}{2} \leq n_2 \leq \frac{L_2-1}{2} \\ 0 & otherwise \end{cases}$. If the window is not centered, $r[n_1, n_2] = \begin{cases} 1 & 0 \leq n_1 < L_1 \\ & \& \ 0 \leq n_2 \leq L_2 \\ 0 & otherwise \end{cases}$. *1st-d-length* gives L_1 and *2nd-d-length* gives L_2. *1st-d-length* and *2nd-d-length* must be integers. If *1st-d-center-p* is centered, *1st-d-length* must be an odd integer. If *2nd-d-center-p* is centered, *2nd-d-length* must be an odd integer. *2nd-d-length* defaults to *1st-d-length*; *1st-d-center-p* defaults to centered; and *2nd-d-center-p* defaults to *1st-d-center-p.*

SINC-SEQUENCE *(length [shift])*
A sinc sequence, $x[n] = \dfrac{\sin(\frac{L}{2}(n+s))}{\frac{L}{2}(n+s)}$ where *length* gives L and *shift* gives s. *length* and *shift* must be real-valued numbers. *shift* defaults to zero.

SINC-SIGNAL *(length)*
A sinc signal, $sinc(e^{j\omega}) = \dfrac{\sin(\frac{L}{2}\omega)}{\frac{L}{2}\sin(\omega)}$ where *length* gives L. *length* must be a real-valued number.

COSINE-SEQUENCE *(frequency)*
A cosine sequence, $x[n] = \cos(\omega_0 n)$ where *frequency* gives ω_0. *frequency* must be a real-valued number.

SINE-SEQUENCE *(frequency)*
A sine sequence, $x[n] = \sin(\omega_0 n)$ where *frequency* gives ω_0. *frequency* must be a real-valued number.

CAUSAL-HAMMING-WINDOW-SEQUENCE *(length)*
A causal, Hamming-window sequence, $h[n] = r_L[n](.54 - .46\cos(\frac{2\pi}{L-1}n))$, where $r_L[n]$ is a causal, rectangular-window sequence and where *length* gives L. *length* must be an integer.

CAUSAL-HANNING-WINDOW-SEQUENCE *(length)*
A causal, Hanning-window sequence, $h[n] = \frac{1}{2}r_L[n](1 - \cos(\frac{2\pi}{L}n))$, where $r_L[n]$ is a causal, rectangular-window sequence and where *length* gives L. *length* must be an integer.

FIR-SEQUENCE *([coeff$_0$ ... coeff$_{N-1}$])*
A FIR sequence, $h[n] = \sum_{i=0}^{N-1} c_i \delta[n-i]$ where *coeff$_i$* gives c_i. *coeff$_0$* ... *coeff$_{N-1}$* must be numbers.

IIR-SEQUENCE *(zt-roc [coeff$_1$ ... coeff$_N$])*
An IIR sequence, $y[n]$, such that the z-transform $Y(z) = \dfrac{1}{1 + \sum_{i=1}^{N} c_i z^{-i}}$ with a region of convergence covering $|z|$ in the interval *zt-roc* where *coeff$_i$* gives c_i. *zt-roc* must be an interval starting at or above zero and *coeff$_1$* ... *coeff$_N$* must be numbers.

CAUSAL-IIR-SEQUENCE *([coeff$_1$ ... coeff$_N$])*
A causal IIR sequence, $y[n]$, such that $y[n] = \delta[n] - \sum_{i=1}^{N} c_i y[n-i]$ and $y[n] = 0$ for $n < 0$ where *coeff$_i$* gives c_i. *coeff$_1$* ... *coeff$_N$* must be numbers.

ANTICAUSAL-IIR-SEQUENCE *([coeff$_1$ ... coeff$_N$])*
A anticausal IIR sequence, $y[n]$, such that $y[n] = \delta[n] - \sum_{i=1}^{N} c_i y[n+i]$ and $y[n] = 0$ for $n > 0$ where *coeff$_i$* gives c_i. *coeff$_1$ ... coeff$_N$* must be numbers.

STABLE-IIR-SEQUENCE *([coeff$_1$ ... coeff$_N$]))*
An IIR sequence, $y[n]$, such that the z transform $Y(z) = \dfrac{1}{1 + \sum_{i=1}^{N} c_i z^{-i}}$ with a region of convergence covering $|z| = 1$ where *coeff$_i$* gives c_i. *coeff$_1$ ... coeff$_N$* must be numbers.

A.1.2 Specific system classes

The system classes currently provided in ADE which can generate specific systems are listed in Table A.2. These classes can not be used to generate abstract systems. Instead, the generalization must occur in their parameter values.

The specific systems in these classes can be created using SPECIFIC-MEMBER as described above. These specific systems can also be created and retrieved using the following functions.

(REAL-FFT *length*) *(sequence)*
Create the L-point discrete Fourier-transform sequence,
$y[k] = \begin{cases} \sum_{n=-\infty}^{\infty} Re\{x[n]\}e^{-j\frac{2\pi}{L}kn} & 0 \leq n < L \\ 0 & otherwise \end{cases}$ using a radix-2 FFT for a real-valued sequence. *length* gives L and *sequence* gives $x[n]$. *length* must be a positive integer of the form 2^{ν} and *sequence* must be a discrete-time sequence.

(COMPLEX-FFT *length*) *(sequence)*
Create the L-point discrete Fourier-transform sequence,
$y[k] = \begin{cases} \sum_{n=-\infty}^{\infty} x[n]e^{-j\frac{2\pi}{L}kn} & 0 \leq n < L \\ 0 & otherwise \end{cases}$ using a radix-2 FFT. *length* gives L and *sequence* gives $x[n]$. *length* must be a positive integer of the form 2^{ν} and *sequence* must be a discrete-time sequence.

(FFT *length*) *(sequence)*
Create the L-point discrete Fourier-transform sequence of $x[n]$, where *length* gives

Table A.2: System classes in ADE containing specific systems

REAL-FFT	SEQUENCE-MULTIPLY
COMPLEX-FFT	SIGNAL-MULTIPLY
FFT	ZT-MULTIPLY
COMPLEX-DFT	2D-MULTIPLY
DFT	MULTIPLY
DISCRETE-FOURIER-TRANSFORM	SEQUENCE-CONVOLVE
REAL-IFFT	SIGNAL-CONVOLVE
COMPLEX-IFFT	ZT-CONVOLVE
IFFT	2D-CONVOLVE
COMPLEX-IDFT	CONVOLVE
IDFT	2D-CONVOLVE-1ST-D
INVERSE-DISCRETE-FOURIER-TRANSFORM	2D-CONVOLVE-2ND-D
SEQUENCE-ALIAS-AND-WINDOW	SEQUENCE-SHIFT
SEQUENCE-CIRCULAR-SHIFT	SIGNAL-SHIFT
SEQUENCE-CIRCULAR-REVERSE	2D-SHIFT
SEQUENCE-CIRCULAR-CONVOLVE	SHIFT
SEQUENCE-CONVOLVE-OVERLAP-SAVE	SEQUENCE-SCALE
BANK-OF-SEQUENCES	SIGNAL-SCALE
ROTATED-BANK-OF-SEQUENCES	ZT-SCALE
SHORT-TIME-WINDOW-SEQUENCE	2D-SCALE
SHORT-TIME-FT	SCALE
MODULATED-FILTER-BANK	2D-SCALE-1ST-D
SEQUENCE-FROM-FUNCTION	2D-SCALE-2ND-D
MAP-OVER	SEQUENCE-RECIPROCAL
FIR-FILTER	SIGNAL-RECIPROCAL
CAUSAL-IIR-FILTER	ZT-RECIPROCAL
ANTICAUSAL-IIR-FILTER	2D-RECIPROCAL
CAUSAL-ALL-PASS-SECTION	RECIPROCAL
IDENTITY-SYSTEM	SEQUENCE-DIVIDE
SEQUENCE-ADD	SIGNAL-DIVIDE
SIGNAL-ADD	ZT-DIVIDE
ZT-ADD	2D-DIVIDE
2D-ADD	DIVIDE
ADD	ZT-CONJUGATE-INPUT

Table A.2 continued

SEQUENCE-SUBTRACT	SEQUENCE-CONJUGATE
SIGNAL-SUBTRACT	SIGNAL-CONJUGATE
ZT-SUBTRACT	ZT-CONJUGATE
2D-SUBTRACT	2D-CONJUGATE
SUBTRACT	COMPLEX-CONJUGATE
SEQUENCE-REAL-PART	2D-ABSOLUTE-VALUE
SIGNAL-REAL-PART	ABSOLUTE-VALUE
ZT-REAL-PART	UPSAMPLE
2D-REAL-PART	INTERLEAVE
REAL-PART	DOWNSAMPLE
SEQUENCE-IMAG-PART	SIGNAL-ALIAS-IN-2PI
SIGNAL-IMAG-PART	SEQUENCE-SCALE-INDEX
ZT-IMAG-PART	SIGNAL-SCALE-INDEX
2D-IMAG-PART	ZT-SCALE-INDEX
IMAG-PART	SCALE-INDEX
SEQUENCE-MAGNITUDE	SEQUENCE-REVERSE
SIGNAL-MAGNITUDE	SIGNAL-REVERSE
ZT-MAGNITUDE	FOURIER-TRANSFORM-SYSTEM
2D-MAGNITUDE	FOURIER-TRANSFORM
MAGNITUDE	INVERSE-FOURIER-TRANSFORM-SYSTEM
SEQUENCE-PHASE	INVERSE-FOURIER-TRANSFORM
SIGNAL-PHASE	Z-TRANSFORM-SYSTEM
ZT-PHASE	Z-TRANSFORM
2D-PHASE	INVERSE-Z-TRANSFORM-SYSTEM
INPUT-PHASE	INVERSE-Z-TRANSFORM
SEQUENCE-ABSOLUTE-VALUE	INVERSE-TRANSFORM
SIGNAL-ABSOLUTE-VALUE	ZT-CONTOUR
ZT-ABSOLUTE-VALUE	SEQUENCE-WINDOW
2D-WINDOW	SIGNAL-WINDOW

L and *sequence* gives $x[n]$. *length* must be a positive integer of the form 2^ν and *sequence* must be a discrete-time sequence. This uses REAL-FFT or COMPLEX-FFT, as appropriate.

(COMPLEX-DFT *length*) *(sequence)*
Create the L-point discrete Fourier-transform sequence,
$$y[k] = \begin{cases} \sum\limits_{n=-\infty}^{\infty} x[n]e^{-j\frac{2\pi}{L}kn} & 0 \le n < L \\ 0 & otherwise \end{cases}$$
, where *length* gives L and *sequence* gives $x[n]$. *length* must be a positive integer and *sequence* must be a discrete-time sequence.

(DFT *length*) *(sequence)*
Create the L-point discrete Fourier-transform sequence of $x[n]$, where *length* gives L and *sequence* gives $x[n]$. *length* must be a positive integer and *sequence* must be a discrete-time sequence. This uses COMPLEX-DFT and is included as a parallel to FFT.

(DISCRETE-FOURIER-TRANSFORM *length*) *(sequence)*
Create the L-point discrete Fourier-transform sequence of $x[n]$, where *length* gives L and *sequence* gives $x[n]$. *length* must be a positive integer and *sequence* must be a discrete-time sequence. This uses FFT or DFT, as appropriate.

(REAL-IFFT *length*) *(sequence)*
Create the L-point inverse discrete Fourier-transform sequence,
$$y[k] = \begin{cases} \frac{1}{L}\sum\limits_{n=-\infty}^{\infty} Re\{X[n]e^{j\frac{2\pi}{L}kn}\} & 0 \le n < L \\ 0 & otherwise \end{cases}$$
using a radix-2 IFFT for a real-valued output sequence. *length* gives L and *sequence* gives $X[n]$. *length* must be a positive integer of the form 2^ν and *sequence* must be a discrete-time sequence.

(COMPLEX-IFFT *length*) *(sequence)*
Create the L-point inverse discrete Fourier-transform sequence,
$$y[k] = \begin{cases} \frac{1}{L}\sum\limits_{n=-\infty}^{\infty} X[n]e^{j\frac{2\pi}{L}kn} & 0 \le n < L \\ 0 & otherwise \end{cases}$$
using a radix-2 FFT. *length* gives L and *sequence* gives $X[n]$. *length* must be a positive integer of the form 2^ν and *sequence* must be a discrete-time sequence.

(IFFT *length*) *(sequence)*
Create the L-point inverse discrete Fourier-transform sequence of $x[n]$ using a radix-2 FFT. *length* gives L and *sequence* gives $X[n]$. *length* must be a positive integer of the form 2^ν and *sequence* must be a discrete-time sequence. This uses REAL-IFFT or COMPLEX-IFFT, as appropriate.

(COMPLEX-IDFT *length*) *(sequence)*
Create the L-point inverse discrete Fourier-transform sequence

$y[k] = \begin{cases} \frac{1}{L} \sum_{n=-\infty}^{\infty} X[n] e^{j\frac{2\pi}{L}kn} & 0 \le n < L \\ 0 & otherwise \end{cases}$, where *length* gives L and *sequence* gives $X[n]$. *length* must be a positive integer and *sequence* must be a discrete-time sequence.

(IDFT *length*) *(sequence)*
Create the L-point inverse discrete Fourier-transform sequence of $x[k]$, where *length* gives L and *sequence* gives $X[n]$. *length* must be a positive integer and *sequence* must be a discrete-time sequence. This uses COMPLEX-IDFT and is included as a parallel to IFFT.

(INVERSE-DISCRETE-FOURIER-TRANSFORM *length*) *(sequence)*
Create the L-point inverse discrete Fourier-transform sequence of $x[k]$, where *length* gives L and *sequence* gives $X[n]$. *length* must be a positive integer and *sequence* must be a discrete-time sequence. This uses IFFT or IDFT, as appropriate.

(SEQUENCE-ALIAS-AND-WINDOW *period*) *(sequence)*
Create the sequence $y[n] = r_P[n] \sum_{k=-\infty}^{\infty} x[n+Pk]$ with $r_P[n]$ as the P-point causal, rectangular-window sequence, where *period* gives P and *sequence* gives $x[n]$. *period* must be a positive integer and *sequence* must be a discrete-time sequence.

(SEQUENCE-CIRCULAR-SHIFT *period shift*) *(sequence)*
Create the sequence $y[n] = r_P[n]x[((n+s) \bmod P)]$ with $r_P[n]$ as the P-point causal, rectangular-window sequence, where *period* gives P, *shift* gives s and *sequence* gives $x[n]$. *period* must be a positive integer, *shift* must be an integer and *sequence* must be a discrete-time sequence.

(SEQUENCE-CIRCULAR-REVERSE *period*) *(sequence)*
Create the sequence $y[n] = r_P[n]x[((-n) \bmod P)]$ with $r_P[n]$ as the P-point causal, rectangular-window sequence, where *period* gives P and *sequence* gives $x[n]$. *period* must be a positive integer and *sequence* must be a discrete-time sequence.

(SEQUENCE-CIRCULAR-CONVOLVE *period*) *([seq$_1$ ... seq$_N$])*
Create the sequence $y[n] = r_P[n](x_1[n] \textcircled{P} \ldots \textcircled{P} x_N[n]$ with $u[n] \textcircled{P} v[n] = \sum_{k=0}^{P} u[k]v[((n-k) \bmod P)]$, where *period* gives P and *seq$_i$* gives $x_i[n]$. *period* must be a positive integer and *seq$_1$ ... seq$_N$* must be discrete-time sequences.

(SEQUENCE-OVERLAP-SAVE-CONVOLVE *impulse-response*) *(sequence)*
Create the sequence $y[n] = h[n]*x[n]$ using the overlap-save method, where *impulse-*

response gives $h[n]$ and *sequence* gives $x[n]$. *impulse-response* and *sequence* must be discrete-time sequences.

BANK-OF-SEQUENCES *([seq_0 ... seq_{N-1}])*
Create the two-dimensional sequence $y[n_1,n_2] = \begin{cases} x_{n_2}[n_1] & 0 \leq i < N \\ 0 & otherwise \end{cases}$ where seq_i gives $x_i[n]$. seq_0 ... seq_{N-1} must be discrete-time sequences.

ROTATED-BANK-OF-SEQUENCES *([seq_1 ... seq_N])*
Create the two-dimensional sequence $y[n_1,n_2] = \begin{cases} x_{n_1}[n_2] & 0 \leq n_1 < N \\ 0 & otherwise \end{cases}$ where seq_i gives $x_i[n]$. seq_0 ... seq_{N-1} must be discrete-time sequences.

(SHORT-TIME-WINDOW-SEQUENCE *window [downsampling-factor]*) *(sequence)*
Create the two-dimensional sequence $y[n_1,n_2] = w[n_2]x[Dn_1+n_2]$ where *downsampling-factor* gives D, *window* gives $w[n]$ and *sequence* gives $x[n]$. *downsampling-factor* must be a positive integer and *window* and *sequence* must be discrete-time sequences. *downsampling-factor* defaults to one.

(SHORT-TIME-FT *window [ft-size] [downsampling-factor]*) *(sequence)*
Create the short-time Fourier transform $y[n_1,n_2] = \sum_{n=-\infty}^{\infty} w[n]x[Dn_1+n]e^{-j\frac{2\pi}{N}kn_2}$ where *ft-size* gives N, *downsampling-factor* gives D, *window* gives $w[n]$ and *sequence* gives $x[n]$. *ft-size* and *downsampling-factor* must be positive integers and *window* and *sequence* must be discrete-time sequences. *ft-size* defaults to (INTERVAL-LENGTH (NON-ZERO-SUPPORT *window*)). *downsampling-factor* defaults to one.

(MODULATED-FILTER-BANK *impulse-response N [downsampling-factor]*) *(sequence)*
Create the two-dimensional sequence $y[n,k] = y_{\uparrow D}[Dn,k]$ with $y_{\uparrow D}[n,k] = (h[n]e^{j\frac{2\pi}{N}kn}) * x[n]$, where *downsampling-factor* gives D, *impulse-response* gives $h[n]$ and *sequence* gives $x[n]$. N and *downsampling-factor* must be positive integers and *impulse-response* and *sequence* must be discrete-time sequences. *downsampling-factor* defaults to one.

SEQUENCE-FROM-FUNCTION *(function [arg_1 ... arg_N])*
Create the sequence $y[n] = f(n, arg_1, ..., arg_N)$ where *function* gives $f(\,)$. *function* must a function.

MAP-OVER *(system index start end pattern)*
Create a signal with a correspondence constraint. The signal is generated by applying *system* to the inputs generated by evaluating *pattern* with *index* bound to values from *start* below *end* (by increments of one). *index* is not evaluated and must be a symbol. *start* and *end* must be real-valued numbers.

(FIR-FILTER *[coeff$_0$ ... coeff$_{N-1}$]*) *(sequence)*
Create the sequence $y[n] = h[n] * x[n]$ where (FIR-SEQUENCE *coeff$_0$ ... coeff$_{N-1}$*) gives $h[n]$ and *sequence* gives $x[n]$. *sequence* must be a discrete-time sequence and *coeff$_0$ ... coeff$_{N-1}$* must be numbers.

(CAUSAL-IIR-FILTER *[coeff$_1$ ... coeff$_N$]*) *(sequence)*
Create the sequence $y[n] = x[n] - \sum_{i=1}^{N} c_i y[n-i]$ with the recursion running causally, where *coeff$_i$* gives c_i and *sequence* gives $x[n]$. *sequence* must be a discrete-time sequence and *coeff$_1$ ... coeff$_N$* must be numbers.

(ANTICAUSAL-IIR-FILTER *[coeff$_1$ ... coeff$_N$]*) *(sequence)*
Create the sequence $y[n] = x[n] - \sum_{i=1}^{N} c_i y[n-i]$ with the recursion running causally, where *coeff$_i$* gives c_i and *sequence* gives $x[n]$. *sequence* must be a discrete-time sequence and *coeff$_1$ ... coeff$_N$* must be numbers.

(CAUSAL-ALL-PASS-SECTION *pole$_1$ ... pole$_N$]*) *(sequence)*
Create the sequence $y[n]$, such that $Y(z) = X(z)\frac{\prod_{i=1}^{N} p_i^* - z^{-1}}{\prod_{i=1}^{N} 1 - p_i z^{-1}}$ with a region of convergence on $|z| > \max\{|p_1|, ..., |p_N|\}$, where *pole$_i$* gives p_i and *sequence* gives $x[n]$. *sequence* must be a discrete-time sequence and *pole$_1$ ... pole$_N$* must be numbers.

IDENTITY-SYSTEM *(input)*
Pass *input* unchanged.

SEQUENCE-ADD *([seq$_1$ ... seq$_N$])*
Create the sequence $y[n] = \sum_{i=1}^{N} x_i[n]$, where *seq$_i$* gives $x_i[n]$. *seq$_1$ ... seq$_N$* must be discrete-time sequences.

SIGNAL-ADD *([ft-sig$_1$ ... ft-sig$_N$])*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = \sum_{i=1}^{N} X_i(e^{j\omega})$, where *ft-sig$_i$* gives $X_i(e^{j\omega})$. *ft-sig$_1$ ... ft-sig$_N$* must be discrete-time Fourier-transform signals.

ZT-ADD *([zt-sig$_1$ ... zt-sig$_N$])*
Create the z-transform signal $Y(z) = \sum_{i=1}^{N} X_i(z)$, where *zt-sig$_i$* gives $X_i(z)$. *zt-sig$_1$ ... zt-sig$_N$* must be z-transform signals.

2D-ADD *([2d-seq$_1$ ... 2d-seq$_N$])*
Create the two-dimensional sequence $y[n_1, n_2] = \sum_{i=1}^{N} x_i[n_1, n_2]$, where *2d-seq$_i$* gives $x_i[n_1, n_2]$. *2d-seq$_1$ ... 2d-seq$_N$* must be two-dimensional sequences.

ADD *([input$_1$ ... input$_N$])*
Create the sum of the inputs. This uses SEQUENCE-ADD, SIGNAL-ADD, ZT-ADD or 2D-ADD, as appropriate.

SEQUENCE-SUBTRACT *(sequence [seq$_1$ ... seq$_N$])*
Create the sequence $y[n] = x[n] - \sum_{i=1}^{N} x_i[n]$, where *sequence* gives $x[n]$ and *seq$_i$* gives $x_i[n]$. *sequence* and *seq$_1$ ... seq$_N$* must be discrete-time sequences.

SIGNAL-SUBTRACT *(ft-signal [ft-sig$_1$ ... ft-sig$_N$])*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = X(e^{j\omega}) - \sum_{i=1}^{N} X_i(e^{j\omega})$, where *ft-signal* gives $X(e^{j\omega})$ and *ft-sig$_i$* gives $X_i(e^{j\omega})$. *ft-signal* and *ft-sig$_1$ ... ft-sig$_N$* must be discrete-time Fourier-transform signals.

ZT-SUBTRACT *(zt-signal [zt-sig$_1$ ... zt-sig$_N$])*
Create the z-transform signal $Y(z) = X(z) - \sum_{i=1}^{N} X_i(z)$, where *zt-signal* gives $X(z)$ and *zt-sig$_i$* gives $X_i(z)$. *zt-signal* and *zt-sig$_1$ ... zt-sig$_N$* must be z-transform signals.

2D-SUBTRACT *(2d-sequence [2d-seq$_1$ ... 2d-seq$_N$])*
Create the two-dimensional sequence $y[n_1, n_2] = x[n_1, n_2] - \sum_{i=1}^{N} x_i[n_1, n_2]$, where *2d-sequence* gives $x[n_1, n_2]$ and *2d-seq$_i$* gives $x_i[n_1, n_2]$. *2d-sequence* and *2d-seq$_1$ ... 2d-seq$_N$* must be two-dimensional sequences.

SUBTRACT *(input [input$_1$ ... input$_N$])*
Create the difference of the inputs. This uses SEQUENCE-SUBTRACT, SIGNAL-SUBTRACT, ZT-SUBTRACT or 2D-SUBTRACT, as appropriate.

SEQUENCE-MULTIPLY *([seq$_1$ ... seq$_N$])*
Create the sequence $y[n] = \prod_{i=1}^{N} x_i[n]$, where *seq$_i$* gives $x_i[n]$. *seq$_1$ ... seq$_N$* must be discrete-time sequences.

SIGNAL-MULTIPLY *([ft-sig$_1$ ... ft-sig$_N$])*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = \prod_{i=1}^{N} X_i(e^{j\omega})$, where *ft-sig$_i$* gives $X_i(e^{j\omega})$. *ft-sig$_1$ ... ft-sig$_N$* must be discrete-time Fourier-transform signals.

ZT-MULTIPLY *([zt-sig$_1$ ... zt-sig$_N$])*
Create the z-transform signal $Y(z) = \prod_{i=1}^{N} X_i(z)$, where *zt-sig$_i$* gives $X_i(z)$. *zt-sig$_1$ ... zt-sig$_N$* must be z-transform signals.

2D-MULTIPLY *([2d-seq$_1$ ... 2d-seq$_N$])*
Create the two-dimensional sequence $y[n_1, n_2] = \prod_{i=1}^{N} x_i[n_1, n_2]$, where *2d-seq$_i$* gives $x_i[n_1, n_2]$. *2d-seq$_1$ ... 2d-seq$_N$* must be two-dimensional sequences.

MULTIPLY *([input$_1$ ... input$_N$])*
Create the product of the inputs. This uses SEQUENCE-MULTIPLY, SIGNAL-MULTIPLY, ZT-MULTIPLY or 2D-MULTIPLY, as appropriate.

(SEQUENCE-WINDOW *interval*) *(sequence)*
Create the sequence $y[n] = \begin{cases} x[n] & n \in \textit{interval} \\ 0 & \textit{otherwise} \end{cases}$, where *sequence* gives $x[n]$. *sequence* must be a discrete-time sequence and *interval* must be an interval.

(SIGNAL-WINDOW *interval*) *(ft-signal)*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = \begin{cases} X(e^{j\omega}) & \omega \in \textit{interval} \\ 0 & \textit{otherwise} \end{cases}$, where *ft-signal* gives $X(e^{j\omega})$. *ft-signal* must be a discrete-time Fourier-transform signal and *interval* must be an interval.

(2D-WINDOW *1st-d-ivl [2nd-d-ivl]*) *(2d-sequence)*
Create the two-dimensional sequence $y[n_1, n_2] = \begin{cases} x[n_1, n_2] & n_1 \in \textit{1st-d-ivl} \\ & \& \; n_2 \in \textit{2nd-d-ivl} \\ 0 & \textit{otherwise} \end{cases}$, where *2d-sequence* gives $x[n_1, n_2]$. *2d-sequence* must be a two-dimensional sequence and *1st-d-ivl* and *2nd-d-ivl* must be intervals. *2nd-d-ivl* defaults to $[-\infty\ \infty]$.

(WINDOW-INPUT *interval [2nd-d-ivl]*) *(input)*
Create the windowed version of the input. This uses SEQUENCE-WINDOW, SIGNAL-WINDOW or 2D-WINDOW, as appropriate.

SEQUENCE-CONVOLVE *([seq$_1$ ... seq$_N$])*
Create the sequence $y[n] = x_1[n] * \ldots * x_N[n]$, where *seq$_i$* gives $x_i[n]$. *seq$_1$ ... seq$_N$* must be discrete-time sequences.

SIGNAL-CONVOLVE *([ft-sig$_1$ ... ft-sig$_N$])*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = X_1(e^{j\omega}) * \ldots * X_N(e^{j\omega})$, where *ft-sig$_i$* gives $X_i(e^{j\omega})$. *ft-sig$_1$ ... ft-sig$_N$* must be discrete-time Fourier-transform signals.

ZT-CONVOLVE *([zt-sig$_1$ ... zt-sig$_N$])*
Create the z-transform signal $Y(z) = X_1(z) * \ldots * X_N(z)$, where *zt-sig$_i$* gives $X_i(z)$. *zt-sig$_1$ ... zt-sig$_N$* must be z-transform signals.

2D-CONVOLVE *([2d-seq$_1$ ... 2d-seq$_N$])*
Create the two-dimensional sequence $y[n_1, n_2] = x_1[n_1, n_2] * ... * x_N[n_1, n_2]$, where *2d-seq$_i$* gives $x_i[n_1, n_2]$. *2d-seq$_1$ ... 2d-seq$_N$* must be two-dimensional sequences.

CONVOLVE *([input$_1$ ... input$_N$])*
Create the convolution of the inputs. This uses SEQUENCE-CONVOLVE, SIGNAL-CONVOLVE, ZT-CONVOLVE or 2D-CONVOLVE, as appropriate.

(SEQUENCE-SHIFT *shift*) *(sequence)*
Create the sequence $y[n] = x[n + s]$, where *sequence* gives $x[n]$ and *shift* gives s. *sequence* must be a discrete-time sequence and *shift* must be an integer.

(SIGNAL-SHIFT *shift*) *(ft-signal)*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = X(e^{j(\omega+s)})$, where *ft-signal* gives $X(e^{j\omega})$ and *shift* gives s. *ft-signal* must be a discrete-time Fourier-transform signal and *shift* must be a real number.

(2D-SHIFT *shift1 [shift2]*) *(2d-sequence)*
Create the two-dimensional sequence $y[n_1, n_2] = x[n_1 + s_1, n_2 + s_2]$, where *2d-sequence* gives $x[n_1, n_2]$, *shift1* gives s_1 and *shift2* gives s_2. *2d-sequence* must be a two-dimensional sequence and *shift1* and *shift2* must be integers. *shift2* defaults to zero.

(SHIFT *shift1 [shift2]*) *(input)*
Create a shifted version of the input. This uses SEQUENCE-SHIFT, SIGNAL-SHIFT or 2D-SHIFT, as appropriate.

(SEQUENCE-SCALE *scale*) *(sequence)*
Create the sequence $y[n] = ax[n]$, where *sequence* gives $x[n]$ and *scale* gives a. *sequence* must be a discrete-time sequence and *scale* must be a number.

(SIGNAL-SCALE *scale*) *(ft-signal)*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = aX(e^{j\omega})$, where *ft-signal* gives $X(e^{j\omega})$ and *scale* gives a. *ft-signal* must be a discrete-time Fourier-transform signal and *scale* must be a number.

(ZT-SCALE *scale*) *(zt-signal)*
Create the z-transform signal $Y(z) = aX(z)$, where *zt-signal* gives $X(z)$ and *scale* gives a. *zt-signal* must be a z-transform signal and *scale* must be a number.

(2D-SCALE *scale*) *(2d-sequence)*
Create the two-dimensional sequence $y[n_1, n_2] = ax[n_1, n_2]$, where *2d-sequence* gives $x[n_1, n_2]$ and *scale* gives a. *2d-sequence* must be a two-dimensional sequence and *scale* must be a number.

(SCALE *scale*) *(input)*
Create a scaled version of the input. This uses SEQUENCE-SCALE, SIGNAL-SCALE, ZT-SCALE or 2D-SCALE, as appropriate.

(2D-SCALE-1ST-D *1d-sequence*) *(2d-sequence)*
Create the two-dimensional sequence $y[n_1, n_2] = a[n_2]x[n_1, n_2]$, where *2d-sequence* gives $x[n_1, n_2]$ and *1d-sequence* gives $a[n]$. *2d-sequence* must be a two-dimensional sequence and *1d-sequence* must be a discrete-time sequence.

(2D-SCALE-2ND-D *1d-sequence*) *(2d-sequence)*
Create the two-dimensional sequence $y[n_1, n_2] = a[n_1]x[n_1, n_2]$, where *2d-sequence* gives $x[n_1, n_2]$ and *1d-sequence* gives $a[n]$. *2d-sequence* must be a two-dimensional sequence and *1d-sequence* must be a discrete-time sequence.

SEQUENCE-RECIPROCAL *(sequence)*
Create the sequence $y[n] = \frac{1}{x[n]}$, where *sequence* gives $x[n]$. *sequence* must be a discrete-time sequence.

SIGNAL-RECIPROCAL *(ft-signal)*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = \frac{1}{X(e^{j\omega})}$, where *ft-signal* gives $X(e^{j\omega})$. *ft-signal* must be a discrete-time Fourier-transform signal.

ZT-RECIPROCAL *(zt-signal)*
Create the z-transform signal $Y(z) = \frac{1}{X(z)}$, where *zt-signal* gives $X(z)$. *zt-signal* must be a z-transform signal.

2D-RECIPROCAL *(2d-sequence)*
Create the two-dimensional sequence $y[n_1, n_2] = \frac{1}{x[n_1, n_2]}$, where *2d-sequence* gives $x[n_1, n_2]$. *2d-sequence* must be a two-dimensional sequence.

RECIPROCAL *(input)*
Create a reciprocal of the input. This uses SEQUENCE-RECIPROCAL, SIGNAL-RECIPROCAL, ZT-RECIPROCAL or 2D-RECIPROCAL, as appropriate.

SEQUENCE-DIVIDE *(sequence [seq$_1$ ... seq$_N$])*
Create the sequence $y[n] = \frac{x[n]}{\prod_{i=1}^{N} x_i[n]}$, where *sequence* gives $x[n]$ and *seq$_i$* gives $x_i[n]$. *sequence* and *seq$_1$ ... seq$_N$* must be discrete-time sequences.

SIGNAL-DIVIDE *(ft-signal [ft-sig$_1$ ... ft-sig$_N$])*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = \frac{X(e^{j\omega})}{\prod_{i=1}^{N} X_i(e^{j\omega})}$, where

ft-signal gives $X(e^{j\omega})$ and *ft-sig$_i$* gives $X_i(e^{j\omega})$. *ft-signal* and *ft-sig$_1$ ... ft-sig$_N$* must be discrete-time Fourier-transform signals.

ZT-DIVIDE *(zt-signal [zt-sig$_1$ ... zt-sig$_N$])*
Create the z-transform signal $Y(z) = \frac{X(z)}{\prod_{i=1}^{N} X_i(z)}$, where *zt-signal* gives $X(z)$ and *zt-sig$_i$* gives $X_i(z)$. *zt-signal* and *zt-sig$_1$ ... zt-sig$_N$* must be z-transform signals.

2D-DIVIDE *(2d-sequence [2d-seq$_1$ ... 2d-seq$_N$])*
Create the two-dimensional sequence $y[n_1, n_2] = \frac{x[n_1, n_2]}{\prod_{i=1}^{N} x_i[n_1, n_2]}$, where *2d-sequence* gives $x[n_1, n_2]$ and *2d-seq$_i$* gives $x_i[n_1, n_2]$. *2d-sequence* and *2d-seq$_1$ ... 2d-seq$_N$* must be two-dimensional sequences.

DIVIDE *(input [divide$_1$ ... divide$_N$])*
Create the difference of the inputs. This uses SEQUENCE-DIVIDE, SIGNAL-DIVIDE, ZT-DIVIDE or 2D-DIVIDE, as appropriate.

SEQUENCE-CONJUGATE *(sequence)*
Create the sequence $y[n] = x^*[n]$, where *sequence* gives $x[n]$. *sequence* must be a discrete-time sequence.

SIGNAL-CONJUGATE *(ft-signal)*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = X^*(e^{j\omega})$, where *ft-signal* gives $X(e^{j\omega})$. *ft-signal* must be a discrete-time Fourier-transform signal.

ZT-CONJUGATE *(zt-signal)*
Create the z-transform signal $Y(z) = X^*(z)$, where *zt-signal* gives $X(z)$. *zt-signal* must be a z-transform signal.

2D-CONJUGATE *(2d-sequence)*
Create the two-dimensional sequence $y[n_1, n_2] = x^*[n_1, n_2]$, where *2d-sequence* gives $x[n_1, n_2]$. *2d-sequence* must be a two-dimensional sequence.

COMPLEX-CONJUGATE *(input)*
Create conjugate of the input. This uses SEQUENCE-CONJUGATE, SIGNAL-CONJUGATE, ZT-CONJUGATE or 2D-CONJUGATE, as appropriate.

ZT-CONJUGATE-INPUT *(zt-signal)*
Create the z-transform signal $Y(z) = X(z^*)$, where *zt-signal* gives $X(z)$. *zt-signal* must be a z-transform signal.

SEQUENCE-REAL-PART *(sequence)*
Create the sequence $y[n] = Re\{x[n]\}$, where *sequence* gives $x[n]$. *sequence* must be a discrete-time sequence.

SIGNAL-REAL-PART *(ft-signal)*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = Re\{X(e^{j\omega})\}$, where *ft-signal* gives $X(e^{j\omega})$. *ft-signal* must be a discrete-time Fourier-transform signal.

ZT-REAL-PART *(zt-signal)*
Create the z-transform signal $Y(z) = Re\{X(z)\}$, where *zt-signal* gives $X(z)$. *zt-signal* must be a z-transform signal.

2D-REAL-PART *(2d-sequence)*
Create the two-dimensional sequence $y[n_1, n_2] = Re\{x[n_1, n_2]\}$, where *2d-sequence* gives $x[n_1, n_2]$. *2d-sequence* must be a two-dimensional sequence.

REAL-PART *(input)*
Take the real part of the input. This uses SEQUENCE-REAL-PART, SIGNAL-REAL-PART, ZT-REAL-PART or 2D-REAL-PART, as appropriate.

SEQUENCE-IMAG-PART *(sequence)*
Create the sequence $y[n] = Im\{x[n]\}$, where *sequence* gives $x[n]$. *sequence* must be a discrete-time sequence.

SIGNAL-IMAG-PART *(ft-signal)*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = Im\{X(e^{j\omega})\}$, where *ft-signal* gives $X(e^{j\omega})$. *ft-signal* must be a discrete-time Fourier-transform signal.

ZT-IMAG-PART *(zt-signal)*
Create the z-transform signal $Y(z) = Im\{X(z)\}$, where *zt-signal* gives $X(z)$. *zt-signal* must be a z-transform signal.

2D-IMAG-PART *(2d-sequence)*
Create the two-dimensional sequence $y[n_1, n_2] = Im\{x[n_1, n_2]\}$, where *2d-sequence* gives $x[n_1, n_2]$. *2d-sequence* must be a two-dimensional sequence.

IMAG-PART *(input)*
Take the imaginary part of the input. This uses SEQUENCE-IMAG-PART, SIGNAL-IMAG-PART, ZT-IMAG-PART or 2D-IMAG-PART, as appropriate.

SEQUENCE-MAGNITUDE *(sequence)*
Create the sequence $y[n] = \|x[n]\|$, where *sequence* gives $x[n]$. *sequence* must be a discrete-time sequence.

SIGNAL-MAGNITUDE *(ft-signal)*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = \|X(e^{j\omega})\|$, where *ft-signal* gives $X(e^{j\omega})$. *ft-signal* must be a discrete-time Fourier-transform signal.

ZT-MAGNITUDE *(zt-signal)*
Create the z-transform signal $Y(z) = \|X(z)\|$, where *zt-signal* gives $X(z)$. *zt-signal* must be a z-transform signal.

2D-MAGNITUDE *(2d-sequence)*
Create the two-dimensional sequence $y[n_1, n_2] = \|x[n_1, n_2]\|$, where *2d-sequence* gives $x[n_1, n_2]$. *2d-sequence* must be a two-dimensional sequence.

MAGNITUDE *(input)*
Take the magnitude of the input. This uses SEQUENCE-MAGNITUDE, SIGNAL-MAGNITUDE, ZT-MAGNITUDE or 2D-MAGNITUDE, as appropriate.

SEQUENCE-PHASE *(sequence)*
Create the sequence $y[n] = \angle\{x[n]\}$, where *sequence* gives $x[n]$. *sequence* must be a discrete-time sequence.

SIGNAL-PHASE *(ft-signal)*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = \angle\{X(e^{j\omega})\}$, where *ft-signal* gives $X(e^{j\omega})$. *ft-signal* must be a discrete-time Fourier-transform signal.

ZT-PHASE *(zt-signal)*
Create the z-transform signal $Y(z) = \angle\{X(z)\}$, where *zt-signal* gives $X(z)$. *zt-signal* must be a z-transform signal.

2D-PHASE *(2d-sequence)*
Create the two-dimensional sequence $y[n_1, n_2] = \angle\{x[n_1, n_2]\}$, where *2d-sequence* gives $x[n_1, n_2]$. *2d-sequence* must be a two-dimensional sequence.

INPUT-PHASE *(input)*
Take the phase of the input. This uses SEQUENCE-PHASE, SIGNAL-PHASE, ZT-PHASE or 2D-PHASE, as appropriate.

SEQUENCE-ABSOLUTE-VALUE *(sequence)*
Create the sequence $y[n] = |x[n]|$, where *sequence* gives $x[n]$. *sequence* must be a real-valued discrete-time sequence.

SIGNAL-ABSOLUTE-VALUE *(ft-signal)*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = |X(e^{j\omega})|$, where *ft-signal* gives $X(e^{j\omega})$. *ft-signal* must be a real-valued discrete-time Fourier-transform signal.

ZT-ABSOLUTE-VALUE *(zt-signal)*
Create the z-transform signal $Y(z) = |X(z)|$, where *zt-signal* gives $X(z)$. *zt-signal* must be a real-valued z-transform signal.

2D-ABSOLUTE-VALUE *(2d-sequence)*
Create the two-dimensional sequence $y[n_1, n_2] = |x[n_1, n_2]|$, where *2d-sequence* gives $x[n_1, n_2]$. *2d-sequence* must be a real-valued two-dimensional sequence.

ABSOLUTE-VALUE *(input)*
Take the absolute value of the input. This uses SEQUENCE-ABSOLUTE-VALUE, SIGNAL-ABSOLUTE-VALUE, ZT-ABSOLUTE-VALUE or 2D-ABSOLUTE-VALUE, as appropriate.

(UPSAMPLE L) *(sequence)*
Create the sequence $y[n] = \begin{cases} x[n/L] & n = Lk \\ 0 & otherwise \end{cases}$, where *sequence* gives $x[n]$. *sequence* must be a discrete-time sequence and L must be a positive integer.

INTERLEAVE *(seq$_0$ [seq$_1$ ... seq$_{N-1}$])*
Create the sequence $y[n] = x_{(n \bmod N)}[\lfloor \frac{n}{N} \rfloor]$, where *seq$_i$* gives $x_i[n]$. *seq$_0$ ... seq$_{N-1}$* must be discrete-time sequences.

(DOWNSAMPLE M) *(sequence)*
Create the sequence $y[n] = x[Mn]$, where *sequence* gives $x[n]$. *sequence* must be a discrete-time sequence and M must be a positive integer.

(SIGNAL-ALIAS-IN-2PI M) *(ft-signal)*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = \sum_{k=0}^{M-1} X(e^{j(\omega + \frac{2\pi}{M}k)})$, where *ft-signal* gives $X(e^{j\omega})$. *ft-signal* must be a discrete-time Fourier-transform signal and M must be a positive integer.

(SEQUENCE-SCALE-INDEX *scale*) *(sequence)*
Create the sequence $y[n] = x[sn]$, where *sequence* gives $x[n]$ and *scale* gives s. *sequence* must be a discrete-time sequence and *scale* must be ± 1.

(SIGNAL-SCALE-INDEX *scale*) *(ft-signal)*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = X(e^{js\omega})$, where *ft-signal* gives $X(e^{j\omega})$ and *scale* gives s. *ft-signal* must be a discrete-time Fourier-transform signal and *scale* must be a real number.

(ZT-SCALE-INDEX *scale*) *(zt-signal)*
Create the z-transform signal $Y(z) = X(sz)$, where *zt-signal* gives $X(z)$ and *scale* gives s. *zt-signal* must be a z-transform signal and *scale* must be a number.

(SCALE-INDEX *scale*) *(input)*
Create the version of the input with a scaled index. This uses SEQUENCE-SCALE-INDEX, SIGNAL-SCALE-INDEX, ZT-SCALE-INDEX or 2D-SCALE-INDEX, as appropriate.

SEQUENCE-REVERSE *(sequence)*
Create the sequence $y[n] = x[-n]$, where *sequence* gives $x[n]$. *sequence* must be a discrete-time sequence.

SIGNAL-REVERSE *(ft-signal)*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = \|X(e^{-j\omega})\|$, where *ft-signal* gives $X(e^{j\omega})$. *ft-signal* must be a discrete-time Fourier-transform signal.

FOURIER-TRANSFORM-SYSTEM *(sequence)*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = \mathcal{F}\{x[n]\}$ where *sequence* gives $x[n]$. *sequence* must be a discrete-time sequence.

FOURIER-TRANSFORM *(sequence)*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = \mathcal{F}\{x[n]\}$ where *sequence* gives $x[n]$. This function uses the value of the property FT. If no closed-form expression can be found, (FOURIER-TRANSFORM-SYSTEM *sequence*) is used. If the Fourier transform is known to be non-existent, #<NONE> is used. *sequence* must be a discrete-time sequence.

INVERSE-FOURIER-TRANSFORM-SYSTEM *(ft-signal)*
Create the discrete-time sequence $y[k] = \mathcal{F}^{-1}\{X(e^{j\omega})\}$ where *ft-signal* gives $X(e^{j\omega})$. *ft-signal* must be a discrete-time Fourier-transform signal.

INVERSE-FOURIER-TRANSFORM *(ft-signal)*
Create the discrete-time sequence $y[n] = \mathcal{F}^{-1}\{X(e^{j\omega})\}$ where *ft-signal* gives $X(e^{j\omega})$. This function uses the value of the property IFT. If no closed-form expression can be found, (INVERSE-FOURIER-TRANSFORM-SYSTEM *ft-signal*) is used. *ft-signal* must be a discrete-time Fourier-transform signal.

Z-TRANSFORM-SYSTEM *(sequence)*
Create the z-transform signal $Y(z) = \mathcal{Z}\{x[n]\}$ where *sequence* gives $x[n]$. *sequence* must be a discrete-time sequence.

Z-TRANSFORM *(sequence)*
Create the z-transform signal $Y(z) = \mathcal{Z}\{x[n]\}$ where *sequence* gives $x[n]$. This function uses the value of the property ZT. If no closed-form expression can be found, (Z-TRANSFORM-SYSTEM *sequence*) is used. If the z transform is known to be non-existent, #<NONE> is used. *sequence* must be a discrete-time sequence.

INVERSE-Z-TRANSFORM-SYSTEM *(zt-signal)*
Create the discrete-time sequence $y[k] = \mathcal{Z}^{-1}\{X(z)\}$ where *zt-signal* gives $X(z)$. *zt-signal* must be a z-transform signal.

INVERSE-Z-TRANSFORM *(zt-signal)*
Create the discrete-time sequence $y[n] = \mathcal{Z}^{-1}\{X(z)\}$ where *zt-signal* gives $X(z)$. This function uses the value of the property IZT. If no closed-form expression can be found, (INVERSE-Z-TRANSFORM-SYSTEM *zt-signal*) is used. *zt-signal* must be a z-transform signal.

INVERSE-TRANSFORM *(input)*
Create the discrete-time sequence which is the inverse transform of the input signal. This function uses INVERSE-FOURIER-TRANSFORM or INVERSE-Z-TRANSFORM, as appropriate. *input* must be a discrete-time Fourier-transform signal or a z-transform signal.

Table A.3: Signal classes in ADE containing abstract, inherent signals

DISCRETE-TIME-SEQUENCE	CONSTANT
FOURIER-DOMAIN-SIGNAL	IMPULSE
Z-DOMAIN-SIGNAL	GENERAL-EXPONENTIAL
2D-SEQUENCE	COMPLEX-EXPONENTIAL
SINC	CAUSAL-RECTANGULAR-WINDOW

(ZT-CONTOUR *[initial-radius] [relative-2pi-radius]*) *(zt-signal)*
Create the discrete-time Fourier-transform signal $Y(e^{j\omega}) = X(z)|_{z=r_0 r^{\omega} e^{j\omega}}$ where *zt-signal* gives $X(z)$, *initial-radius* gives r_0 and *relative-2pi-radius* gives r. *zt-signal* must be a z-transform signal with explicitly known pole and zero polynomials. *initial-radius* defaults to one. *relative-2pi-radius* defaults to one.

A.1.3 Abstract, inherent signal classes

The inherent signal classes currently provided in ADE which can generate abstract signals are listed in Table A.3. Any of these classes can be used for the value of *class* when using A-MEMBER-OF, as described above. None of these signal classes can be used to generate a specific signal, since their definition does not provide a unique signal description.

A.1.4 Abstract system classes

The system classes currently provided in ADE which can generate abstract systems are listed in Table A.4. Any of these classes can be used for the value of *class* when using A-MEMBER-OF, as described above. None of these system classes can be used to generate a specific system, since their definition does not provide a unique system description.

Table A.4: System classes in ADE containing abstract systems

DISCRETE-TIME-SYSTEM	2D-CONSTANT-IN-1D
2D-SYSTEM	ADD-SYSTEM
FOURIER-DOMAIN-SYSTEM	SUBTRACT-SYSTEM
Z-DOMAIN-SYSTEM	MULTIPLY-SYSTEM
SHIFT-INVARIANT-SYSTEM	CONVOLVE-SYSTEM
GENERALIZED-SHIFT-INVARIANT-SYSTEM	SHIFT-SYSTEM
ROTATED-SHIFT-INVARIANT-SYSTEM	SCALE-SYSTEM
GENERALIZED-ROTATED-SHIFT-INVARIANT-SYSTEM	RECIPROCAL-SYSTEM
COMMUTATIVE-ASSOCIATIVE-SYSTEM	CONJUGATE-SYSTEM
MEMORYLESS-SYSTEM	REAL-PART-SYSTEM
ASSOCIATIVE-SYSTEM	IMAG-PART-SYSTEM
ADDITIVE-SYSTEM	MAGNITUDE-SYSTEM
HOMOGENEOUS-SYSTEM	PHASE-SYSTEM
GENERALIZED-HOMOGENEOUS-SYSTEM	ABSOLUTE-VALUE-SYSTEM
LINEAR-SYSTEM	SCALE-INDEX-SYSTEM
LINEAR-SUPPORT-SENSITIVE-SYSTEM	REVERSE-INPUT
GENERALIZED-LINEAR-SYSTEM	FT-SYSTEM
DFT-SYSTEM	IFT-SYSTEM
FFT-SYSTEM	ZT-SYSTEM
IDFT-SYSTEM	IZT-SYSTEM
IFFT-SYSTEM	

A.1.5 Retrieval of property values

Signals and systems are characterized by explicit, observable properties, such as system linearity and signal support. Some of these properties, like signal domain and system linearity, are defined by using the signal and system class hierarchy. The remainder of the currently defined signal and system properties are listed here.

INVERTIBLE-P: T or NIL.
Whether or not the system is invertible.

INVERSE-SYSTEM: a system or #<UNKNOWN>
The inverse system, if it exists.

SAMPLE-TYPE: 'EXTENDED-NUMBER, 'EXTENDED-REAL-NUMBER, 'EXTENDED-IMAGINARY-NUMBER or any non-numeric data type.
The type of the sample values of the signal.

RANGE: a range of values described by the ranges of real and imaginary parts of the sample values.
The range of the sample values of the signal.

NON-ZERO-SUPPORT: any interval.
The interval on which the signal is not identically zero.

PERIODICITY: any non-negative number.
The periodicity of the signal.

SYMMETRY: any symmetry descriptor.
A detailed description of the symmetry characteristics of the signal.

COMPUTABLE-P: T or NIL.
Whether or not the sample values of the signal are all computable.

SAMPLES-COMPUTABLE-P: T or NIL.
Whether or not the sample values of the signal are computable individually.

FT: any discrete-time Fourier-transform signal.
The Fourier transform of the discrete-time sequence.

BANDWIDTH: any real number between 0 and 2π.
The bandwidth of the discrete-time sequence.

FREQUENCY-SUPPORT: any interval within $\{-\pi \ \pi\}$.
The frequency support of the discrete-time sequence.

IFT: any discrete-time sequence.
The inverse Fourier transform of the discrete-time Fourier-transform signal.

ZT: any z-transform signal.
The z transform of the discrete-time sequence.

IZT: any discrete-time sequence.
The inverse z transform of the z-transform signal.

ROC: any interval of radii covered by $\{0\ \infty\}$, the null interval or #<UNKNOWN>.
The region of convergence of the z-transform signal.

POLES: any polynomial or #<UNKNOWN>.
The polynomial, $D(z)$, such that with the zeroes polynomial, $N(z)$, the z-transform signal is described by the ratio $\frac{N(z)}{D(z)}$.

ZEROES: any polynomial or #<UNKNOWN>. The polynomial, $N(z)$, such that with the poles polynomial, $D(z)$, the z-transform signal is described by the ratio $\frac{N(z)}{D(z)}$.

COST: any cost descriptor.
A detailed description of the cost of computing the sample values of the signal.

EQUIVALENT-FORMS: a list of equivalent signals or systems.
The expressions which are equivalent to the signal or system.

CONSTRAINED-EQUIVALENT-FORMS: a list of equivalent signals or systems.
The expressions which are equivalent to the signal or system and which maintain all correspondence constraints.

EFFICIENT-IMPLEMENTATIONS: a list of equivalent signals or systems.
The equivalent forms of the signal or system which are not dominated by any of the other equivalent forms.

CONSTRAINED-EFFICIENT-IMPLEMENTATIONS: a list of equivalent signals or systems.
The constrained equivalent forms of the signal or system which are not dominated by any of the other constrained equivalent forms.

SIMPLIFICATION: the original or a simpler signal or system.
The simplest description of the signal or system which can be found directly.

CONSTRAINED-SIMPLIFICATION: the original or a simpler signal or system.
The simplest description of the signal or system which can be found directly and which maintains all correspondence constraints.

EFFECTIVE-FORMS: the original signal or a signal independent of transform systems.
The description of the signal with all Fourier and z transform pairs removed.

A.1.6 Retrieval of sample values

ADE provides a uniform interface for retrieving sample values from signals, largely independent of the identity of the signal and its programming model. As in SPLICE (Dove et al., 1984; Myers, 1986), two distinct mechanisms are provided for retrieving sample values from signals.

FETCH *(signal index)*
The sample value of the signal at *index*. *signal* must be a discrete-time sequence, a discrete-time Fourier-transform signal, a z-transform signal or a two-dimensional sequence. If signal is a discrete-time sequence, *index* must be an integer. If signal is a discrete-time Fourier-transform signal, *index* must be a real number. If signal is a z-transform signal, *index* must be a number within the region of convergence. If signal is a a two-dimensional sequence, *index* must be an integer. The "sample value" which is returned in this case is the two-dimensional sequence which represents $x[n_1, n_2]|_{n_1=index}$.

FETCH-INTERVAL *(signal interval [output-array] [sample-spacing])*
An array of the sample values of the signal within *interval* with the sampling grid size indicated by *sample-spacing*. *interval* must be an interval and, if given, *output-array* must be a one-dimensional array with the correct number of elements to hold the sample values. *signal* must be a discrete-time sequence, a discrete-time Fourier-transform signal, a z-transform signal or a two-dimensional sequence. If signal is a discrete-time sequence, *interval* must have an integer starting point and *sample-spacing* must be an integer. If signal is a discrete-time Fourier-transform signal, *interval* must have an real-valued starting point and *sample-spacing* must be a real number. If signal is a z-transform signal, all the samples indicated by *interval* and *sample-spacing* must be the region of convergence. If signal is a two-dimensional sequence, *interval* must have an integer starting point and *sample-spacing* must be an integer. The array of "sample values" which is returned in this case contains the one-dimensional sequences, $x[n_1, n_2]|_{n_1=index}$ for each index. *sample-spacing* defaults to one.

Four related functions are also provided for retrieving sample values:

FETCH-REAL *(signal index)*
The real part of the sample value of the signal at *index*.

FETCH-IMAG *(signal index)*
The imaginary part of the sample value of the signal at *index*.

FETCH-INTERVAL-REAL *(signal interval [output-array] [sample-spacing])*
An array of the real parts of the sample values of the signal at the indicated indices.

FETCH-INTERVAL-IMAG *(signal interval [output-array] [sample-spacing])*
An array of the imaginary parts of the sample values of the signal at the indicated indices.

FETCH, FETCH-INTERVAL and the associated real- and imaginary-part functions can be used to retrieve sample values from anywhere in the domain of a signal, independent of the non-zero support. They can be applied indiscriminately to discrete-time sequences, discrete-time Fourier-transform signals and z-transform signals; to abstract signals, simple specific signals and symbolically constrained signals; to computable and uncomputable signals. The only restriction which the identity of the signal places on the use of FETCH and FETCH-INTERVAL is the restriction that the sample-value requests truly lie within the domain of the signal. For discrete-time sequences, this translates into the constraint that all the sample value requests fall on integer indices. For z-transform signals, this translates into the constraint that all the sample value requests fall within the region of convergence. As in SPLICE (Dove et al., 1984; Myers, 1986), the external behavior of the interface is independent of the particular programming paradigm used in calculating the sample values, allowing random access to the sample values.

A.2 Functions for Creating and Manipulating Intervals, Symbolic Numbers and Polynomials

A.2.1 Interval representation and manipulation

Intervals are used in ADE to describe sets of numbers. Examples of their use include the description of the non-zero support of a discrete-time sequence, the non-zero support of a discrete-time Fourier-transform signal and the range of a real-valued signal. These examples include two distinct types of intervals: the non-zero support of a discrete-time sequence is an interval containing only integers while the other two intervals contain all the numbers lying between the end points of the interval. The interval of integers [*start end*] contains represents the set of integers, n, such that *start* $\leq n <$ *end*. For discrete intervals, the starting and ending points must be either integers or real numbers. If no integers lie in the interval, then a unique empty interval is returned. The continuous interval $\{$*start end*$\}$ represents the set of numbers, z, such that $Im(z) = Im(start) = Im(end)$ and $Re(start) \leq Re(z) < Re(end)$. From this description of continuous intervals, the starting and ending points can be complex numbers, as long as their imaginary parts are equal. If $Re(start) > Re(end)$, then no numbers will lie in the interval and the unique empty interval is returned. Finally, the interval $\{$*point point*$\}$ is used to represent the continuous interval containing only the single number, *point*. Functions for accessing and manipulating intervals are described here.

NULL-INTERVAL
The variable bound to the unique empty interval.

INTERVAL *(start end [continuous-p])*
Creates an interval from the given specification. If *continuous-p* is NIL, the interval is discrete; otherwise, the interval is continuous.

INTERVAL-P *(object)*
Whether or not *object* is an interval.

INTERVAL-START *(interval)*
The lowest point in the interval.

INTERVAL-REAL-START *(interval)*
The real-part of the lowest point in the interval.

INTERVAL-END *(interval)*
The point just beyond the end of the interval. If the interval is continuous, the end point is the value which was given for *end* in creating the interval. If the interval is discrete, the end point is the integer which is greater than or equal to the value which was given for *end* in creating the interval.

INTERVAL-REAL-END *(ivl)*
The real-part of the end of the interval.

INTERVAL-LENGTH *(interval)*
The length of the interval.

INTERVAL-LAST *(ivl)*
The last point in the interval. If the interval is continuous, the last point is the value which was given for *end* in creating the interval. If the interval is discrete, the last point is the integer which is less than the value which was given for *end* in creating the interval.

INTERVAL-REAL-LAST *(ivl)*
The real-part of the last point in the interval.

INTERVAL-IMAGPART *(ivl)*
The imaginary part of the numbers within the interval.

INTERVAL-CONTINUOUS-P *(ivl)*
Whether or not *object* is a continuous interval.

NON-EMPTY-INTERVAL-P *(interval)*
Whether or not *interval* has one or more points within it.

INFINITE-INTERVAL-P *(interval)*
Whether or not *interval* is an infinite-length interval.

FINITE-INTERVAL-P *(interval)*
Whether or not *interval* is a finite-length interval. This function assumes that the empty interval is not a finite-length interval.

NULL-INTERVAL-P *(interval)*
Whether or not *interval* is the empty interval.

POINT-INTERVAL-P *(interval)*
Whether or not *interval* contains one single point.

INTERVAL-EQ *(ivl1 ivl2)*
Whether or not the intervals are the same.

WITH-STACK-INTERVAL *((var start end [continuous-p]) body)*
Evaluates *body* with *var* bound to the interval (INTERVAL START END [CONTINUOUS-P])

INTERVAL-INTERSECT *([ivl_1 ... ivl_N])*
The intersection of ivl_1 ... ivl_N

INTERVAL-ADJOINING-P *([ivl_1 ... ivl_N])*
Whether or not the intervals ivl_1 ... ivl_N jointly cover (INTERVAL-COVER ivl_1 ... ivl_N)

INTERVAL-COVER *([ivl$_1$ ... ivl$_N$])*
The interval which covers of *ivl$_1$ ... ivl$_N$*.

INTERVAL-COVERS-P *(ivl1 ivl2)*
Whether or not *ivl1* covers *ivl2.*

INTERVAL-INTERSECT-P *([ivl$_1$ ... ivl$_N$])*
Whether or not the intervals *ivl$_1$ ... ivl$_N$* all intersect.

INTERVAL-ADVANCE *(interval advance)*
The interval with the starting and ending points of *interval* increased by *advance.*

INTERVAL-RETARD *(interval retard)*
The interval with the starting and ending points of *interval* decreased by *retard.*

INTERVAL-REVERSE *(interval)*
The interval which is *interval* flipped about zero.

INTERVAL-CONVOLVE *(ivl$_1$ [ivl$_2$ ... ivl$_N$])*
The interval which represents the non-zero support of a convolution output when the inputs have the non-zero supports *ivl$_1$ ... ivl$_N$*.

INTERVAL-CORRELATE *(ivl1 ivl2)*
The interval which represents the non-zero support of a correlation between two signals with the supports *ivl1* and *ivl2.*

INTERVAL-AUTOCORRELATE *(interval)*
The interval which represents the non-zero support of an autocorrelation of a signal with the support *interval.*

INTERVAL-CONVOLUTION-SUPPORT *(filter-ivl requested-ivl)*
The interval over which a signal must be known to compute the requested convolution interval using the given filter interval.

VALID-CONVOLUTION-INTERVAL *(sequence-ivl filter-ivl)*
The interval over which there is full overlap of the *sequence-ivl* and *filter-ivl* (no edge effects).

ORIGINAL-SUPPORT *(filtered-ivl filter-ivl)*
The original non-zero support of the signal given the post-convolution support and the support of the filter.

CENTERED-INTERVAL *(length [continuous-p])*
The interval of length *length* which is centered about zero. If the interval is discrete, *length* must be an odd integer. *continuous-p* defaults to NIL.

INTERVAL-ALIAS *(from-ivl to-ivl)*
The list of pairs of intervals which represent the result of circularly aliasing *from-ivl* into *to-ivl*

WITH-INTERVAL-ALIAS *((from-var from-ivl to-var to-ivl) body)*
Evaluates *body* once for each of the pairs of intervals representing the result of circularly aliasing of *from-ivl* into *to-ivl*. During each evaluation, *from-var* is bound to the current section of *from-ivl* and *to-var* is bound to the corresponding section of *to-ivl*

$GET-INTERVAL *(object)*
An interval which in some way describes *object*.

INTERVAL-COMPLEMENT *(interval universe)*
The intervals which are the parts of *universe* and which are outside of *interval*

A.2.2 Representation and manipulation of symbolic numbers

Symbolic numbers are generally required to describe abstract signals and symbolically constrained signals and systems. New symbolic numbers of a given type and range can be generated by using the format:

A-MEMBER-OF *(numeric-type [&properties relation*$_1$ *value*$_1$ ... *relation*$_N$ *value*$_N$*])*
A symbolic number, *symbolic-number*, of the type *numeric-type* which satisfies the relations *symbolic-number relation*$_1$ *value*$_1$, ..., *symbolic-number relation*$_N$ *value*$_N$.

Constraints can also be imposed on a symbolic number indirectly by a constraint on a related symbolic number. An example of this is provided by constraining the magnitude of a symbolic, complex number to be less than one: no constraints are placed directly on the symbolic, complex number yet the range of its possible values is constrained. ADE attempts to propagate these constraints internally.

A variety of extended algebraic and trigonometric functions are provided in ADE for manipulating numbers, symbolic numbers, $-\infty$ and ∞ uniformly. When confronted with symbolic numbers, the majority of these will produce another symbolic number to represent their output. $<, $≤, $=, $≥ and $> attempt to determine a boolean value by

Table A.5: Functions for manipulating symbolic numbers in ADE

INFINITE-NUMBER-P *(number)*	\$+ *(num$_1$ [num$_2$ ... num$_n$])*
INFINITE-REAL-NUMBER-P *(number)*	\$* *(num$_1$ [num$_2$ ... num$_n$])*
INFINITE-IMAG-NUMBER-P *(number)*	\$− *(num$_1$ [num$_2$ ... num$_n$])*
\$REALPART *(number)*	\$/ *(num$_1$ [num$_2$ ... num$_n$])*
\$IMAGPART *(number)*	\$1/ *(number)*
\$CONJUGATE *(number)*	\$1− *(number)*
\$MAGNITUDE *(number)*	\$1+ *(number)*
\$PHASE *(number)*	\$LCM *(num$_1$ [num$_2$ ... num$_n$])*
\$ABS *(real-number)*	\$GCD *(num$_1$ [num$_2$ ... num$_n$])*
\$SIGNUM *(real-number)*	\$MOD *(x y)*
\$NUMERATOR *(real-number)*	\$FLOOR *(x [y])*
\$DENOMINATOR *(real-number)*	\$CEILING *(x [y])*
\$> *(num$_1$ [num$_2$ ... num$_n$])*	\$ROUND *(x [y])*
\$< *(num$_1$ [num$_2$ ... num$_n$])*	\$EXPT *(base-number power-number)*
\$>= *(num$_1$ [num$_2$ ... num$_n$])*	\$EXP *(power)*
\$<= *(num$_1$ [num$_2$ ... num$_n$])*	\$LOG *(number)*
\$MAX *(num$_1$ [num$_2$ ... num$_n$])*	\$COS *(number)*
\$MIN *(num$_1$ [num$_2$ ... num$_n$])*	\$SIN *(number)*
\$MINUS *(number)*	

propagating the constraints on the symbolic numbers. Failing this, these functions will query the user for the required information. The currently available functions for manipulating symbolic numbers are listed in Table A.5. Each simply extends the corresponding algebraic or trigonometric function.

A.2.3 Polynomial representation and manipulation

Polynomials are useful for the description of rational z-transform signals. In these cases, $X(z)$ can be completely specified using $N(z) = \sum_{i=0}^{M} a_i z^i$, $D(z) = \sum_{i=0}^{L} b_i z^i$ and a continuous interval representing the radial extent of the region of convergence. Once such a description of $X(z)$ is found, finding and manipulating the poles, zeroes and region of convergence is greatly simplified. The functions for creating and manipulating single-variable, finite-order polynomials in ADE are described here.

CREATE-POLYNOMIAL *([:COEFFICIENTS coefficients] [:GAIN gain] [:FACTORS factors])*
The polynomial fitting the given specification.

POLYNOMIAL-COEFFICIENTS *(polynomial)*
The list of the coefficients of the polynomial.

POLYNOMIAL-NTH-COEFFICIENT *(n polynomial)*
The coefficient of the n'th power in *polynomial.*

POLYNOMIAL-GAIN *(polynomial)*
The gain of *polynomial.*

POLYNOMIAL-FACTORS *(polynomial)*
The list of pairs, containing the location of the polynomial zeroes and their multiplicity.

POLYNOMIAL-EXPRESSION *(polynomial [variable])*
The MACSYMA expression representing the given polynomial, in powers of *variable. variable* defaults to $z.

POLYNOMIAL-DERIVATIVE *(polynomial)*
The polynomial which is the derivative of the given polynomial.

POLYNOMIAL-MULTIPLY *(poly1 poly2)*
The product of these two polynomials.

POLYNOMIAL-DIVIDE (POLY1 POLY2)
The quotient polynomial and the remainder polynomial from this division. The order of the remainder polynomial will be less than that of *poly2.*

POLYNOMIAL-REMOVE-ZEROES-AT-ZERO *(polynomial)*
The same polynomial with any zeroes at zero removed.

POLYNOMIAL-CONJUGATE *(polynomial)*
The polynomial with the conjugate coefficients.

POLYNOMIAL-REMOVE-COMMON-ZEROES *(poly1 poly2)*
The two polynomials with common factors removed.

POLYNOMIAL-COMMON-ZEROES-P *(poly1 poly2)*
If there are common factors, the polynomial formed from these common factors; otherwise, NIL.

POLYNOMIAL-SUBTRACT *(poly1 poly2)*
The difference polynomial.

POLYNOMIAL-ADD *(poly1 poly2)*
The sum of these two polynomials.

EVAL-POLYNOMIAL-AT *(location numerator-polynomial [denominator-polynomial])* The value of the ratio of the numerator and denominator polynomials when evaluated at *location. denominator-polynomial* defaults to one.

POLYNOMIAL-ORDER *(polynomial)*
The order of *polynomial.*

POLYNOMIAL-SCALE *(scale polynomial)*
The scaled polynomial.

POLYNOMIAL-SCALE-VARIABLE *(scale polynomial)*
The polynomial resulting from scaling the variable.

UNITY-GAIN-POLYNOMIAL *(polynomial)*
The polynomial with the same factors and unity gain.

POLYNOMIAL-DOWNSAMPLE *(m zeroes poles)*
The zeroes polynomial and poles polynomial representing the result of downsampling the sequence described in the z domain by the polynomials *zeroes* and *poles.*

A.3 Functions for Adding New Properties and New Control Strategies

A.3.1 Property declaration

Properties are used to explicitly characterize signals or systems. Often, this characterization only makes sense for a particular set of signals or systems: for example, the property describing the inverse Fourier transform should not be retrievable from a discrete-time sequence. To allow for this type of object sensitive, ADE requires the explicit declaration of properties with the applicable signal and system classes included. The currently declared properties were described in Section A.1.5. New properties can be defined using DECLARE-PROPERTY, as described here.

```
(DEFINE-PROPERTY name
   applicable-classes
   [:BASIC-ASPECT basic-aspect]
   :SEED seed
   :COMBINING-FUNCTION combining-function
   :DEFAULT-VALUE default-value)
```

applicable-classes provides a list of the types of signals and systems to which the property is applicable.

basic-aspect indicates whether or not to include an instance variable in the representations of these signals and systems, corresponding to this particular property: *basic-aspect* defaults to including the instance variable.

seed provides the initial value in any search to determine the value of the property: the expression given by *seed* is evaluated with SELF bound to the object under consideration and the result of the evaluation is used as the initial value.

combining-function is used to combine partial values found in the course of this search. The combining function can also be used to terminate the search. When a new piece of information is found, *combining-function* will be applied two arguments: the new information and the current value as determined by the search so far. This application is expected to return two values: the first value should be the result of combining the new information with the previous information and the second value is used to indicate when the search can be prematurely terminated.

default-value gives a default for the searches in which no information about the property is found. *default-value* is evaluated with SELF bound to the object under consideration and the result will used as the value of the property.

A.3.2 Control strategy definition

Control strategies are useful for providing additional information about the search for property values. They can encode information about effective approaches to the search or they can also be used to increase the accuracy of the determined value. New strategies can be defined using DEFINE-STRATEGY, as described here.

```
(DEFINE-STRATEGY name
   :GOAL matching-pattern
   [:WITH local-variable-forms]
   [:WHEN requirements]
[:REMOVE-STRATEGIES strategy-names]
[:REPLACEMENT-GOAL replacement-goal]    or    [:SUBGOALS subgoal-pairs]
[:DO actions]
[:ASSERT assertions]
[:ANSWER result]    or    [:SET-ANSWER result]
[:DONE]
[:WHEN-DONE]
[:WHEN additional-requirements]
[:SUBGOALS additional-subgoal-pairs]
[:DO additional-actions]
[:ASSERT additional-assertions]
[:ANSWER final-result]    or    [:SET-ANSWER final-result])
```

matching-pattern is of the form (VALUE-OF *object property value*). The matching pattern indicates the type of search which the strategy is encoded to affect. This pattern is used to restrict the properties and objects on which the strategy is tested. If the matching pattern does not match the current goal, the strategy is not considered further. If the pattern does match the current goal, the bindings from this match are enforced in all the remaining parts of the strategy.

local-variable-forms is a list of *local-variable-form*'s and a *local-variable-form* is either (*variable expression*) to bind *variable* to the result of evaluating *expression* in the binding environment or *variable* to include a variable in the binding environment for subsequent use, typically in *subgoal-pairs.* These variables are available for use within the remainder of the strategy.

requirements can provide an arbitrary test expression. If this expression returns a null value when evaluated, the strategy is disqualified and not considered further. Otherwise, the strategy is taken to be applicable to the current goal.

strategy-names is a list of strategies which should not be considered by this goal. The listed strategies are then bypassed by the current goal and by its replacement, if a replacement goal is provided.

replacement-goal can provide a replacement for the current goal. Providing a replacement goal has the effect of removing the current goal from consideration; of bypassing consideration of the remaining strategies by the current goal; of bypassing consideration of any property rules by the current goal; and of removing any subgoals of the current goal or its subordinates from the schedule. Once the replacement goal is complete, the result of the replacement goal becomes the binding for the result variable.

subgoal-pairs is a list of *subgoal-pair*'s and a *subgoal-pair* is (*variable subgoal-description*) to bind *variable* to the result from the subgoal described by *subgoal-description.*

These can be used to compute values upon which the result depends. The results of the subgoals give rise to more variable bindings. Further processing of this particular strategy is suspended until all of the subgoals are completed.

result can provide a result from the strategy. The strategy is allowed to indicate if this answer should be considered as the complete value for the goal, by using :SET-ANSWER. Otherwise, this answer is used as a partial value and is combined with the current value of the goal.

:DONE can be used to explicitly terminate the search. This has the effect of bypassing consideration of the remaining strategies; of bypassing consideration of any property rules; and of removing any subgoals of the current goal or its subordinates from the schedule.

:WHEN-DONE can be used to give the strategy access to the goal when it is exhausted. A goal is described as exhausted when all rules have been considered and all its subgoals are complete. If this access is requested, the result value of the goal is bound to the result variable from the matching pattern of the strategy. The forms in the strategy definition which follow :WHEN-DONE are evaluated in this environment of the exhausted goal

A.4 Functions for Adding New Signal and System Classes

A.4.1 Definition of abstract signal and system classes

Abstract signal and system classes are only able to create abstract objects. This is due to the generality of their description: these classes do contain enough information to provide unique descriptions of their elements. Examples of abstract signal and system classes include DISCRETE-TIME-SEQUENCE and LINEAR-SYSTEM. The currently defined abstract system and inherent signal classes were described in Sections A.1.3 and A.1.4. New abstract system and inherent signal classes can be defined using DEFINE-ABSTRACT-SYSTEM-CLASS and DEFINE-ABSTRACT-SIGNAL-CLASS, as described here.

DEFINE-ABSTRACT-SIGNAL-CLASS

```
(DEFINE-ABSTRACT-SIGNAL-CLASS name parameter-list
      superclasses
  [documentation-string]
  shared-definitions)
```

defines the abstract signal class *name*, with the parameters named in *parameter-list*. This new signal class is a specialization of the signal classes listed in *superclasses*. All the signals produced by the signal class *name* or by any of its subclasses can use the information inherited from the superclasses as well as the information provided by *shared-definitions*.

shared-definitions provide information about the signals in the class *name* and a *shared-definition* is either

- a *shared-method* having the form (*message-name parameter-list body*),
- a *shared-backward-rule* having the form

  ```
  (GOAL property
    :NAME rule-name
    :OBJECT object-pattern
    [:WITH local-variable-forms]
    [:WHEN requirements]
    [:SUBGOALS subgoal-pairs]
    [:DO actions]
    [:ASSERT assertions]
    [:ANSWER result]   or   [:SET-ANSWER result]
    [:DONE])
  ```

  where the matching pattern of the backward-chaining rule will have the form (VALUE-OF *object-pattern property* ?VALUE) and where the remaining parts of the backward-chaining rule are similar to those described for strategies.

- or a *shared-forward-rule* having the form

  ```
  (ASSERTION property
    :NAME rule-name
    :OBJECT object-pattern
    [:WITH local-variable-forms]
    [:WHEN requirements]
    [:DO actions]
    [:ASSERT assertions])
  ```

where the matching pattern of the forward-chaining rule will have the form (VALUE-OF *object-pattern property* ?VALUE) and where the remaining parts of the forward-chaining rule are similar to those described for strategies.

DEFINE-ABSTRACT-SYSTEM-CLASS

```
(DEFINE-ABSTRACT-SYSTEM-CLASS
    (name system-parameters) *
      system-superclasses
  ([documentation-string]
    shared-system-definitions)
  defined-signal-class)
```

```
(DEFINE-ABSTRACT-SYSTEM-CLASS
    (name system-parameters)
    system-input-list
      system-superclasses
  ([documentation-string]
    shared-system-definitions)
  NIL signal-superclasses
  ([documentation-string]
    shared-signal-definitions))
```

defines the abstract system class *name*, with the system parameters named in *system-parameter-list.* This new system class is a specialization of the system classes listed in *system-superclasses.* All the systems produced by the system class *name* or by any of its subclasses can use the information inherited from these superclasses as well as the information provided by *shared-system-definitions.* Each *shared-system-definition* can be either a *shared-method*, a *shared-backward-rule* or a *shared-forward-rule*, as described above.

The system class *name* can use a previously defined signal class *defined-signal-class* as its output signal class. If a previously defined signal class is used as the output signal class, no additional information is supplied about the output signal class in the system-class definition.

More commonly, DEFINE-ABSTRACT-SYSTEM-CLASS defines a new signal class and uses it as the output signal class. In this case, the second syntax shown above is used. In addition to the system class *name*, a signal class named by appending "-OUTPUT" to *name* is defined. A signal in the output signal class is generated by applying a system from the class *name* to a set of inputs corresponding to the arguments of *system-input-list.* This new signal class is a specialization both of the signal classes listed in *signal-superclasses* and of the output signal classes of the system classes *system-superclasses. shared-signal-definitions* provide information about the output signals. Each *shared-signal-definition* can be either a *shared-method*, a *shared-backward-rule* or a *shared-forward-rule*, as described above.

A.4.2 Definition of specific signal and system classes

Specific signal and system classes are only contain specific objects. This is due to the specificity of their description: any generalization of a description within these classes must occur within the parameters to which they are applied. The currently defined specific system and inherent signal classes were described in Sections A.1.1 and A.1.2. New specific system and inherent signal classes can be defined the forms described here.

DEFINE-SIGNAL-CLASS

```
(DEFINE-SIGNAL-CLASS name parameter-list
      superclasses
  [documentation-string]
  shared-definitions)
```

defines the specific signal class *name*, with the parameters named in *parameter-list.* This new signal class is a specialization of the signal classes listed in *superclasses.* All the signals produced by the signal class *name* can use the information inherited from the superclasses as well as the information provided by *shared-definitions.* Each *shared-definition* can be either a *shared-method*, a *shared-backward-rule* or a *shared-forward-rule*, as described above.

DEFINE-SYSTEM-CLASS

```
(DEFINE-SYSTEM-CLASS (name system-parameters) system-input-list
    system-superclasses
  ([documentation-string]
    shared-system-definitions)
  NIL signal-superclasses
  ([documentation-string]
    shared-signal-definitions))
```

defines the specific system class *name*, with the system parameters named in *system-parameter-list.* This new system class is a specialization of the system classes listed in *system-superclasses.* All the systems produced by the system class *name* can use the information inherited from these superclasses as well as the information provided by *shared-system-definitions.* Each *shared-system-definition* can be either a *shared-method*, a *shared-backward-rule* or a *shared-forward-rule*, as described above.

A new signal class is also defined to act as the output signal class. The output signal class name is derived from *name* by appending "-OUTPUT". This new signal class is a subclass of the output signal classes of *system-superclasses* as well as of the classes *additional-output-signal-superclasses*. *shared-signal-definitions* provide information about the output signals in this class, each being either a *shared-method*, a *shared-backward-rule* or a *shared-forward-rule*, as described above.

DEFINE-SIGNAL

```
(DEFINE-SIGNAL name
      superclasses
  [documentation-string]
  definitions)
```

defines the specific signal *name*. This new signal is a member of the signal classes listed in *superclasses*. The signal can use the information inherited from the superclasses as well as the information provided by *definitions*. Each *definition* can be either a *method*, a *backward-rule* or a *forward-rule*, as described above.

DEFINE-SYSTEM

```
(DEFINE-SYSTEM name system-input-list
    system-superclasses
  ([documentation-string]
    system-definitions)
  NIL signal-superclasses
  ([documentation-string]
    shared-signal-definitions))
```

defines the specific system *name*, with the system inputs named in *system-input-list*. This new system is a member of the system classes listed in *system-superclasses*. The system can use the information inherited from these superclasses as well as the information provided by *system-definitions*. Each *system-definition* can be either a *method*, a *backward-rule* or a *forward-rule*, as described above.

A new signal class is also defined to act as the output signal class. The output signal class name is derived from *name* by appending "-OUTPUT". This new signal class is a subclass of the output signal classes of *system-superclasses* as well as of the classes *additional-output-signal-superclasses*. *shared-signal-definitions* provide information about the output signals in this class, each being either a *shared-method*, a *shared-backward-rule* or a *shared-forward-rule*, as described above.

DEFINE-SIGNAL-CLASS-ALIAS

```
(DEFINE-SIGNAL-CLASS-ALIAS name parameter-list
      superclasses
  [documentation-string]
  master-signal
  shared-definitions)
```

defines the specific signal class *name*, with the parameters named in *parameter-list.* The signals in this class are composition operators: that is, they depend on the signal resulting from evaluating *master-signal* in the environment provided by the parameter bindings to provide some or all of their observable characteristics. This new signal class is a specialization of the signal classes listed in *superclasses.* All the signals produced by the signal class *name* can use the information inherited from the superclasses as well as the information provided by *shared-definitions.* Each *shared-definition* can be either a *shared-method*, a *shared-backward-rule* or a *shared-forward-rule*, as described above.

DEFINE-SYSTEM-CLASS-ALIAS

```
(DEFINE-SYSTEM-CLASS-ALIAS
    (name system-parameters) *
      system-superclasses
  ([documentation-string]
     master-system
  shared-system-definitions))
```

```
(DEFINE-SYSTEM-CLASS-ALIAS
    (name system-parameters)
    system-input-list
      system-superclasses
  ([documentation-string]
    SELF
  shared-system-definitions)
  NIL signal-superclasses
  ([documentation-string]
    master-signal
  shared-signal-definitions))
```

defines the specific system class *name*, with the system parameters named in *system-parameter-list.* This new system class is a specialization of the system classes listed in *system-superclasses.* All the systems produced by the system class *name* can use the information inherited from these superclasses as well as the information provided by *shared-system-definitions.* Each *shared-system-definition* can be either a *shared-method*, a *shared-backward-rule* or a *shared-forward-rule*, as described above.

The systems in the system class *name* can be composition operators: that is, they may depend on the system resulting from evaluating *master-system* in the environment provided by the parameter bindings to provide some or all of their observable characteristics and to provide the output signals. In this case, no additional information can be supplied about the output signals in the system-class definition.

More commonly, the output signals are composition operators: that is, they depend on the signal resulting from evaluating *master-signal* in the environment provided by the parameter and input bindings to provide some or all of their observable characteristics. In this case, the second syntax shown above is used. In addition to the system class *name*, a signal class named by appending "-OUTPUT" to *name* is defined. A signal in the output signal class is generated by applying a system from the class *name* to a set of inputs corresponding to the arguments of *system-input-list.* This new signal class is a specialization both of the signal classes listed in *signal-superclasses* and of the output signal classes of the system classes *system-superclasses.* *shared-signal-definitions* provide information about the output signals. Each *shared-signal-definition* can be either a *shared-method*, a *shared-backward-rule* or a *shared-forward-rule*, as described above.

DEFINE-SYSTEM-ALIAS

```
(DEFINE-SYSTEM-ALIAS (name system-input-list
      system-superclasses
   ([documentation-string]
      system-definitions)
   NIL signal-superclasses
   ([documentation-string]
      master-signal
   shared-signal-definitions))
```

defines the specific system *name*, with the system inputs named in *system-input-list.* This new system is a member of the system classes listed in *system-superclasses.* The system can use the information inherited from these superclasses as well as the information provided by *system-definitions.* Each *system-definition* can be either a *method*, a *backward-rule* or a *forward-rule*, as described above.

A new signal class is also defined to act as the output signal class. The output signal class name is derived from *name* by appending "-OUTPUT". The output signals in this class are composition operators: that is, they depend on the signal resulting from evaluating *master-signal* in the environment provided by the parameter and input bindings to provide some or all of their observable characteristics. This new signal class is a subclass of the output signal classes of *system-superclasses* as well as of the classes *additional-output-signal-superclasses.* *shared-signal-definitions* provide information about the output signals in this class, each being either a *shared-method*, a *shared-backward-rule* or a *shared-forward-rule*, as described above.

A.4.3 Sample-value descriptions

If any of the members of a signal class or an output signal class have computable samples, then an algorithmic specification of the signals must either be among the functional forms inherited from generalizations of the class or be among the functional forms included in the class definition. The four programming paradigms developed in SPLICE (Dove et al., 1984; Myers, 1986) are provided for these algorithmic specifications: the point-operator model, the array-operator model, the state-machine model and the composition model. In the point-operator model, a signal is described by a function that computes a single sample value at a time while in the array-operator model a signal is described by a function that computes many sample values simultaneously. The state-machine model describes discrete-time sequences by maintaining state variables and computing the sample values at successive time indices from some given starting index and starting state. In the composition model, a signal is described using a combination of other systems. Signals and systems using the composition model are defined using DEFINE-SIGNAL-CLASS-ALIAS, DEFINE-SYSTEM-CLASS-ALIAS or DEFINE-SYSTEM-ALIAS, which were described in the previous section. The defining forms for the point-operator model, the array-operator model and the state-machine model are described here. These forms are to be included in the signal-class definitions provided for inherent signals or output signals.

Point-operator model

In the point-operator model, a signal is described by a function that computes a single sample value at a time. SAMPLE-VALUE or SAMPLE-VALUE-REAL and SAMPLE-VALUE-IMAG can be used to encode this function. This programming model is appropriate for discrete-time sequences, discrete-time Fourier-transform signals and z-transform signals.

```
(SAMPLE-VALUE (INDEX)
  (applicable-interval-1 sample-value-expression-1)
                  ⋮
  (applicable-interval-N sample-value-expression-N)
  [(:DEFAULT sample-value-default-expression)])

(SAMPLE-VALUE-REAL (INDEX)
  (applicable-interval-1 sample-value-expression-1)
                  ⋮
  (applicable-interval-N sample-value-expression-N)
  [(:DEFAULT sample-value-default-expression)])

(SAMPLE-VALUE-IMAG (INDEX)
  (applicable-interval-1 sample-value-expression-1)
                  ⋮
```

(*applicable-interval-N sample-value-expression-N*)
[(:DEFAULT *sample-value-default-expression*)])

sample-value-expression-i describes the sample values for the indices within the interval *applicable-interval-i.* If a default is given, this expression will be used to compute the sample values of indices within the non-zero support which do not fall within any of the given intervals.

Array-operator model

In the array-operator model, a signal is described by a function that maps intervals of indices into arrays of sample values. INTERVAL-VALUES or INTERVAL-VALUES-REAL and INTERVAL-VALUES-IMAGINARY can be used to encode this function. COMPUTE-INTERVAL, when passed an interval of indices on which the sample values are desired, is used to expand the extent that interval to an interval appropriate for the array-operator encoded by the INTERVAL-VALUES functions. This programming model is appropriate for discrete-time sequences.

(COMPUTE-INTERVAL (DESIRED-INTERVAL) *body*)

returns the interval of sample values which should be requested simultaneously in order both to accommodate the array-operator model and to retrieve the desired interval of values

(INTERVAL-VALUES (INTERVAL OUTPUT-ARRAY)
(*applicable-interval-1 sample-value-expression-1*)
⋮
(*applicable-interval-N sample-value-expression-N*)
[(:DEFAULT *sample-value-default-expression*)])

(INTERVAL-VALUES-REAL (INTERVAL OUTPUT-ARRAY)
(*applicable-interval-1 sample-value-expression-1*)
⋮
(*applicable-interval-N sample-value-expression-N*)
[(:DEFAULT *sample-value-default-expression*)])

(INTERVAL-VALUE-IMAG (INTERVAL OUTPUT-ARRAY)
(*applicable-interval-1 sample-value-expression-1*)

⋮

(*applicable-interval-N sample-value-expression-N*)
[(:DEFAULT *sample-value-default-expression*)])

sample-value-expression-i places the sample values for the indices within the interval *applicable-interval-i* in the output array given by the parameter OUTPUT-ARRAY. If a default is given, this expression will be used to compute the sample values of indices within the non-zero support which do not fall within any of the given intervals.

State-machine model

The state-machine model describes discrete-time sequences by maintaining state variables and computing the sample values at consecutive indices from the given boundary conditions. The boundary conditions for the state machine are encoded using SM-BOUNDARY and BOUNDARY-STATE: SM-BOUNDARY must determine the boundary index, n_0, and BOUNDARY-STATE must determine $s[n_0]$, the value of the state variable at the boundary. CURRENT-VALUE, when provided with n and the state $s[n]$, must determine $y[n]$, the output value of the state machine at that index. Either NEXT-STATE or PREVIOUS-STATE can be used to complete the description the state-machine. If both NEXT-STATE and PREVIOUS-STATE are included in the state-machine model, the non-zero support is not restricted by the state-machine model. If only NEXT-STATE is included in the state-machine model, the non-zero support is restricted to lie within the interval $[n_0\ \infty]$. Alternately, if only PREVIOUS-STATE is included in the state-machine model, the non-zero support is restricted to lie within the interval $[-\infty\ n_0]$. This programming model is only appropriate for discrete-time sequences.

(SM-BOUNDARY () *body*)

the index of the boundary condition for the state machine

(BOUNDARY-STATE (BOUNDARY-INDEX) *body*)

the boundary condition for the state machine

(PREVIOUS-STATE (CURRENT-INDEX CURRENT-STATE) *body*) the previous state, $s[n-1]$, given the current index, n, and the current state, $s[n]$. This description will only be used to compute the state variables at indices below the boundary index.

(NEXT-STATE (CURRENT-INDEX CURRENT-STATE) *body*)

the next state, $s[n+1]$, given the current index, n, and the current state, $s[n]$. This description will only be used to compute the state variables at indices above the boundary index.

(CURRENT-VALUE (CURRENT-INDEX CURRENT-STATE) *body*)

describes the sample value at the current index, given that index and the current state.

Appendix B

Caching Table Organization

The ideas of unique signal representation and caching were advocated by Kopec (1980), Dove et al. (1984) and Myers (1986) as solutions to two distinct objectives: unique representation was introduced to more closely simulate the mathematical behavior of signals while caching of sample values was introduced to increase computational efficiency. Following these recommendations, caching tables have been used for the output values of signals and systems: systems and inherent signals are cached according to their system or signal class and their parameter values; output signals are cached in their generating system under the system inputs; and sample values are cached in their parent signals under their index.

Caching tables should exploit any restrictions on the indexing keys to increase their efficiency. For example, the indexing domain of a signal restricts the set of possible keys and provides a natural organization to the sample values. Dove et al. (1984) and Myers (1986) exploited this organization to replace general caching tables with arrays when caching signal sample values. In SPLICE, the sample values of discrete-time sequences are stored in arrays with an explicit interval support. The environment insures that new requests for sample values and previously buffered sample values are contiguous by extending the requested interval to cover any intervening regions. The environment also insures that requests for previously computed sample values are serviced using the cached values as opposed to passing these requests onto the function for computing sample values.

The exclusive use of arrays for caching sample values is not suited either for discrete-time Fourier-transform signals or for z-transform signals. Nor can it support requests for sample values at a symbolic index, such as $x[N]$. Finally, in some cases, the extra computation involved in coercing the cache of sample values to be contiguous can be excessive: for example, if a sequence is being downsampled by a large factor, requesting the sample values of the downsampled sequence would incur the overhead of computing not only the sample values which pass through the downsampler but all intervening sample values of the original sequence as well.

To support the diverse requirements on the various caching tables without complicating the interface to the cache unduly, data abstraction is again be used: a uniform set of creation, modification and accessor functions are defined across a variety of internal implementations of the caching table. The cache is then selected to exploit the restrictions on the set of valid keys. The remainder of this subsection is devoted to a description of various caching tables used in ADE.

The most general caching table makes no assumptions about the form of the indexing key. Tables for caching signals and systems are of this type, since there is no natural ordering to these sets of objects. Examples of possible underlying implementations for a general cache include hash tables and association lists.

The most restrictive caching table expects a countable, ordered set of keys. Tables for caching sample values of discrete-time sequences are of this type. The general organization of these caches is shown in Figure B-1. Internally, the cache is supported using two distinct subtables: an internally separate subtable handles requests for sample values at symbolic indices. By separating the cache into two subtables, one for symbolic indices and one for non-symbolic indices, the subtable for non-symbolic indices can exploit both the ordering and the countability of the integer indices. In particular, this subtable can be implemented using: an explicit interval support and a single array completely filled with a contiguous set of sample values (Figure B-2-a); a set of arrays, each with an explicit

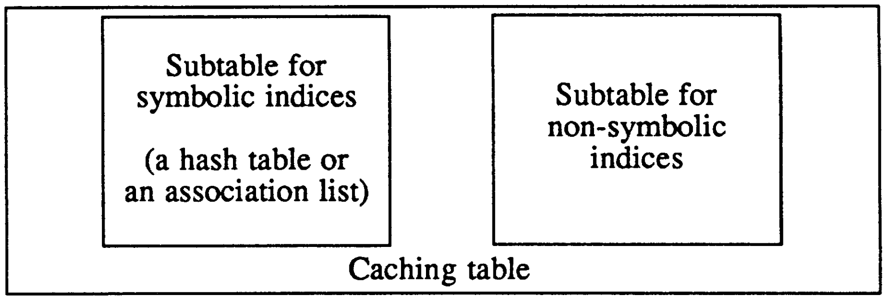
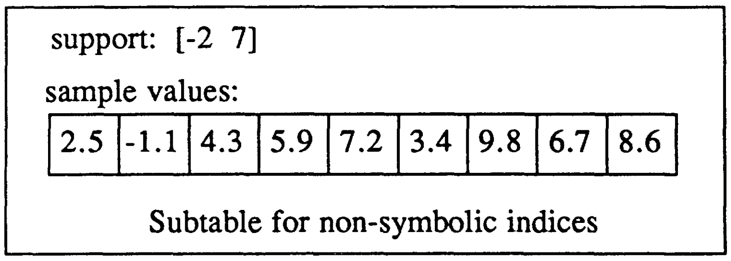

Subtable for symbolic indices
(a hash table or an association list)
Subtable for non-symbolic indices
Caching table

Figure B-1: Proposed internal structure for caching tables designed for ordered sets of keys

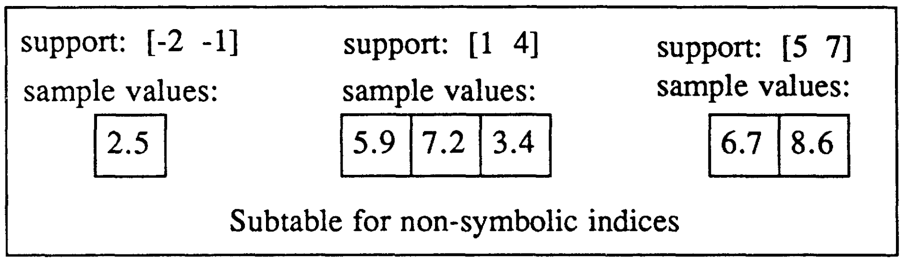
Tables for caching sample values can generally expect an ordered set of keys. However, additional provisions need to be made to handle requests for sample values at symbolic indices. The organization of the cache tables used in ADE is shown here. Internally, the cache uses two distinct subtables: one general subtable to record the sample values at symbolic indices and another to record the sample values at specific numeric indices. By separating the caching table, the subtable for sample values at non-symbolic indices can exploit the ordering of the indices. The interface to the cache obscures these implementational details.

interval support, covering non-overlapping, non-contiguous intervals (Figure B-2-b); or an explicit interval support and a single array, partially filled with sample values with the remaining locations flagged by a unique marker (Figure B-2-c).

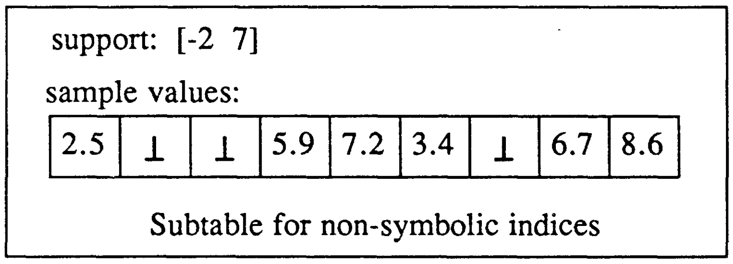
Between these two extremes in caching tables lies one that expects an uncountable, ordered set of keys. Tables for caching sample values of discrete-time Fourier-transform signals are of this type. Tables for caching sample values of z-transform signals are also of this type, using an ordering function which treats the imaginary part of the complex number as being more significant than the real part. Provisions similar to those described in the preceding paragraph have been made for symbolic indices: the layout of the cache uses the same format illustrated in Figure B-1. The subtables used for caching discrete-time Fourier-transform and z-transform sample values at non-symbolic indices do not require countability but do exploit the ordering. Examples of such data structures include binary trees and heaps.



a. the subtable for non-symbolic indices can be implemented using an explicit interval support and a single array, with the array being completely filled with the contiguous set of sample values falling within the support



b. the subtable for non-symbolic indices can be implemented using multiple pairs of an explicit interval support and an array, each pair behaving as described in part (a).



c. the subtable for non-symbolic indices can be implemented using an explicit interval support and a single array, with the array being partially filled with sample values and the remaining locations being flagged by a unique marker.

Figure B-2: Alternate implementations of the subtables for the non-symbolic indices in countable, ordered key sets.

Using these representations of a cache, the structure of the indexing domain is exploited while still maintaining a uniform interface. The rewards of organizing the caching tables as described are twofold. In general, the full caching table of entries need not be considered: in the case of the array-based representations, simply examining the explicit support or supports suffices to determine whether or not an entry is present;[1] in the case of uncountable, ordered representations, a small number of comparisons can be used to determine the same information. Furthermore, in array-based representations, the locality of neighboring samples speeds up responses to requests for intervals of sample values: the sample values that are present have already been grouped and ordered, so that these operations need not be repeated.

[1]If a partially filled array is used as a cache, the value stored in a single location in the array will also need to be checked to insure that the value has been computed.

Appendix C

Pattern Matching in ADE

The first thing considered in testing any rule or strategy is the matching pattern or patterns. Thus, the process of matching is pervasive and important. For a strategy or rule to be applicable, the current goal must match the matching pattern of the strategy or backward-chaining rule: that is, $G \subseteq R$ where G is the current goal and R is the matching pattern of the strategy or rule. For a forward-chaining rule to be applicable, the assertion must match the matching pattern of the rule: that is, $A \subseteq R$ where A is the assertion and R is the matching pattern of the rule. The description of the matching process given here centers on the components of the matching patterns of the rule: thus, the matching behavior will be described from the viewpoint of B in the process $A \subseteq B$.

- $*$: "$*$" matches anything and generates no new bindings.
- a symbol: Any symbol other than "$*$" can only be matched by itself.
- a signal or system: A signal or system can only be matched by itself; by its master, if it is a composition operator; or by a composition operator using it as a master.
- a number or a symbolic number: A number or a symbolic number can only be matched by another number or symbolic number to which it is known to be equal, using previously imposed constraints.
- *?name@type* or *?name@type$subpattern* or *?name$subpattern*, a matching variable:

- A matching variable can be matched by another variable, if the other variable does not have a subpattern; and any previous bindings for either variable matches the other variable and, when both variables have previous bindings, the previous bindings match one another.
- A matching variable can be matched by any object, if the object matches any previous binding for the variable; and the object is of type *type*; and, when the variable has a subpattern, the object matches the subpattern.

- *(mp_1 mp_2 ...)*, a simple list of matching subpatterns:
  - A list can be matched by a matching variable, if the list matches any previous binding for the variable; and the list conforms to the variable type; and, when the variable has a subpattern, the list matches the subpattern.
  - A list can be matched by another list of equal length, if all of the corresponding elements of the lists match, while sharing a single set of bindings.
  - A list can be matched by a specific, inherent signal or a specific system if the list is at least two elements long; and, using a single set of bindings, SPECIFIC-MEMBER or A-MEMBER-OF matches mp_1; and mp_2 matches the type or a supertype of the signal or system; and the list of parameters of the signal or system matches the remaining sublist from the list.
  - A list can be matched by an abstract signal or system if the list is at least two elements long; and, using a single set of bindings, A-MEMBER-OF matches mp_1; and mp_2 matches the type or a supertype of the signal or system; and the list of parameters of the signal or system matches the remaining sublist from the list.
  - A list can be matched by a generated signal if the list is at least two elements long; and, using a single set of bindings, OUTPUT-OF matches mp_1; and mp_2 matches the type or a supertype of the signal or system; and the list of parameters of the signal or system matches the remaining sublist from the list.

- *(mp_1 ... mp_{N-1} &REST mp_N)*, a keyed list of matching subpatterns:
  - A keyed list can be matched by a matching variable, if the keyed list matches any previous binding for the variable; and the variable imposes no type restrictions other than an optional restriction to lists; and the variable does not have a subpattern.
  - A keyed list can be matched by a simple list if the simple list is of length greater than or equal to *N-1*; and, using a single set of bindings, all of the first *N-1* corresponding elements of the lists match; and the remaining sublist from the simple list matches the matching pattern mp_N.

  - A keyed list can be matched by a specific, inherent signal or a specific system if $N > 2$; and, using a single set of bindings, SPECIFIC-MEMBER or A-MEMBER-OF matches mp_1; and mp_2 matches the type or a supertype of the signal or system; and the the list of parameters of the signal or system matches the remaining sublist from the keyed list.
  - A keyed list can be matched by an abstract signal or system if $N > 2$; and, using a single set of bindings, A-MEMBER-OF matches mp_1; and mp_2 matches the type or a supertype of the signal or system; and the the list of parameters of the signal or system matches the remaining sublist from the keyed list.
  - A keyed list can be matched by a generated signal if $N > 2$; and, using a single set of bindings, OUTPUT-OF matches mp_1; and mp_2 matches the type or a supertype of the signal or system; and the the list of parameters of the signal or system matches the remaining sublist from the keyed list.

- *?name@type${element-subpattern}* or *?name${element-subpattern}*, a list-matching variable with an element pattern:

  - A list-matching variable with an element pattern can be matched by another variable, if the other variable does not have a subpattern; and, when there is a previous binding for the other variable, the bound value is a list; and any previous bindings for either variable matches the other variable; and, when both variables have previous bindings, the previous bindings match one another.
  - A list-matching variable with an element pattern can be matched by a list, if the list matches any previous binding for the variable; and the list is of type *type*; and the elements of the list can be matched the subpattern. When the subpattern itself contains variables, the element matches can have distinct bindings for the subpattern variables which are non-parallel component variables but they must share identical bindings for all other subpattern variables.

- *?[name]* or *?[name@type]*, a non-parallel component variable: The matching behavior for a non-parallel component variable is the same as for a simple variable. Non-parallel component variables are used within the subpatterns of parallel list-matching variables to allow non-parallel bindings. The non-parallel component variable is ultimately bound to the list of the bindings from the individual component matches. The type restriction is imposed on the individual component matches, not on this final list.

Appendix D

Dominance relations between cost measures in ADE

As described in Chapter 7, the cost of computing the system inputs is included in the cost of the output signal and, as a result, cost structures must at some point be added. If corresponding cost intervals in two cost structures which are being added have the same indexing block size, the addition is straightforward: corresponding cost vectors are simply added. When corresponding cost intervals do not have equal block sizes, the addition is more involved. The approach taken in ADE to adding corresponding cost intervals with unequal block sizes is to maintain separate cost intervals in the output cost structures for each of the block sizes. Unfortunately, this approach further complicates the determination of dominance between cost structures. The approach that is actually used in ADE in determining dominance is described here.

Dominance between cost structures is determined by comparing the cost intervals in the two cost structures which have the same block sizes and cover the same intervals. Given two cost structures, the cost intervals can be broken up in such a way that each cost interval with a particular grouping in either cost structure is associated with a corresponding cost interval with the same grouping in the other cost structure. In particular, if the corresponding cost interval with the same grouping is not present in the second cost structure, a zero-cost cost interval with the correct grouping can be created. Thus, the pair of cost structures is made up of two sets of corresponding indexing intervals where each indexing interval has associated with it a number of cost intervals with different groupings. The pairs of indexing intervals also correspond on each of the groupings. Within a single indexing interval, $interval_i$, the groupings are numbered from the smallest grouping, as $grouping_{i,1}$ up to the largest grouping, $grouping_{i,N_i}$. Each grouping in either cost structure has associated with it a cost vector: for $grouping_{i,j}$, the cost vector $vector1_{i,j}$ will be the cost of $grouping_{i,j}$ on $interval_i$ in the first cost structure and the cost vector $vector2_{i,j}$ will be the cost of $grouping_{i,j}$ on $interval_i$ in the second cost structure.

Using these notations, dominance is determined using the following recipe:

- Set both 1-DOMINATED-BY-2 and 2-DOMINATED-BY-1 to false.
- For i indexing across all the indexing intervals, $interval_i$
  - Set both EXCESS-COST-IN-1 and EXCESS-COST-IN-2 to the zero-cost vector.
  - For j indexing from 1 through N_i across all the groupings for $interval_i$, $grouping_{i,j}$
    - If both EXCESS-COST-IN-1 and EXCESS-COST-IN-2 are the zero-cost vector,
      - If $vector1_{i,j}$ dominates $vector2_{i,j}$, set 1-DOMINATES-2 to true and set EXCESS-COST-IN-2 to the difference between $vector2_{i,j}$ and $vector1_{i,j}$.
      - Otherwise, if $vector2_{i,j}$ dominates $vector1_{i,j}$, set 2-DOMINATES-1 to true and set EXCESS-COST-IN-1 to the difference between $vector1_{i,j}$ and $vector2_{i,j}$.
      - Otherwise, if $vector1_{i,j}$ and $vector2_{i,j}$ are not equal, set both 1-DOMINATES-2 and 2-DOMINATES-1 to true.
    - Otherwise, if EXCESS-COST-IN-2 is the zero-cost vector
      - If $vector1_{i,j}$ dominates $vector2_{i,j}+$ EXCESS-COST-IN-2, set 1-DOMINATES-2 to true and set EXCESS-COST-IN-2 to the difference between $vector2_{i,j}+$ EXCESS-COST-IN-2 and $vector1_{i,j}$.
      - Otherwise, set both 1-DOMINATES-2 and 2-DOMINATES-1 to true.
    - Otherwise, if EXCESS-COST-IN-1 is the zero-cost vector
      - If $vector2_{i,j}$ dominates $vector1_{i,j}+$ EXCESS-COST-IN-1, set 2-DOMINATES-1 to true and set EXCESS-COST-IN-1 to the difference between $vector1_{i,j}+$ EXCESS-COST-IN-1 and $vector2_{i,j}$.
      - Otherwise, set both 1-DOMINATES-2 and 2-DOMINATES-1 to true.

        * If both 1-DOMINATES-2 and 2-DOMINATES-1 are true, neither cost structure can dominate, so terminate signalling this condition.

      (end loop of j across all the groupings of $interval_i$)

  (end loop of i across all the intervals)

- If both 1-DOMINATES-2 and 2-DOMINATES-1 are true, neither cost structure dominates.
- Otherwise, if 1-DOMINATES-2 is true, the first cost structure dominates.
- Otherwise, if 2-DOMINATES-1 is true, the second cost structure dominates.
- Otherwise, neither cost structure dominates.

Bibliography

Abelson, H. and Sussman, G. J., *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.

Barnwell, T. P., Hodges, C. J. M., and Randolf, M., "Optimal Implementation of Single Time Index Flow Graphs on Synchronous Multiprocessor Machines," In *Proceedings ICASSP '82*, pp. 679–682, 1982.

Barnwell, T. P. and Schwartz, D. A., "Optimal Implementation of Flow Graphs on Synchronous Multiprocessors," In *Proceedings Asilomar Conference on Circuits and Systems*, pp. 188–194, 1983.

Bentz, B., "An Automatic Programming System for Signal Processing Applications," *Pattern Recognition*, 18(6):491–495, 1985.

Chan, D. S. K. and Rabiner, L., "Theory of Roundoff Noise in Cascade Realizations of Finite Impulse Response Digital Filters," *Bell System Technical Journal*, 50(3):329–345, March 1973a.

Chan, D. S. K. and Rabiner, L., "An Algorithm for Minimizing Roundoff Noise in Cascade Realizations of Finite Impulse Response Digital Filters," *Bell System Technical Journal*, 50(3):347–385, March 1973b.

Chefitz, E. L., *Automated Design of Signal-Processing Systems based on Qualitative Signal Descriptions*, Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1987.

Crochiere, R. E. and Rabiner, L. R., "Optimum FIR Digital Filter Implementations for Decimation, Interpolation, and Narrow-Band Filtering," *IEEE Trans. Acoustics, Speech, and Signal Processing*, 23(5):444–456, October 1975.

Dove, W. P., Myers, C. S., and Milios, E. E., *An Object-Oriented Signal Processing Environment: The Knowledge-Based Signal Processing Package*, R.L.E. Technical Report 502, Massachusetts Institute of Technology, Cambridge, MA, 1984.

Dove, W. P., *Knowledge-based Pitch Detection*, PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1986.

DSP Committee, IEEE Acoustics, Speech and Signal Processing Society, *Programs for Digital Signal Processing*, IEEE Press, New York, NY, 1979.

Fikes, R. E. and Nilsson, N. J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence*, 2(3/4):189–208, Winter 1971.

Fogg, D. C., *Assisting Design Given Performance Specifications Involving Multiple Criteria*, PhD proposal, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA, 1988.

Gethöffer, H., Hoffman, K., Lenzer, A., Roethe, N., and Waldschmidt, H., "A Design and Computing System for Signal Processing Applications," In *Proceedings ICASSP '79*, pp. 688–691, 1979.

Gethöffer, H., "SIPROL: A High Level Language for Digital Signal Processing," In *Proceedings ICASSP '80*, pp. 1056–1059, 1980.

Henke, W., *MITSYN - An Interactive Dialogue Language for Time Signal Processing*, R.L.E. Technical Report TM-1, Massachusetts Institute of Technology, Cambridge, MA, 1975.

Hicks, J. E., Jr., *A High-Level Signal Processing Programming Language*, L.C.S. Technical Report 414, Massachusetts Institute of Technology, Cambridge, MA, 1988.

Hsiao, C.-C., "Polyphase Filter Matrix for Rational Sampling Rate Conversions," In *Proceedings ICASSP '87*, pp. 50.4.1–50.4.4, 1987.

Jackson, L. B., "On the Interaction of Roundoff Noise and Dynamic Range in Digital Filters," *Bell System Technical Journal*, 49(2):159–184, February 1970.

Jackson, L. B., *Digital Filters and Signal Processing*, Kluwer Academic Publishers, Hingham, MA, 1986.

Jaffe, J. S., and Richardson, J. M., "A Code-Division Multiple Beam Imaging System," *OCEANS '89*, to be published 1989.

Johnson, D. H., "Signal Processing Software Tools," In *Proceedings ICASSP '84*, pp. 8.6.1–8.6.3, 1984.

Kelly, J. L., Jr., Lochbaum, C., and Vyssotsky, V. A., "A Block Diagram Compiler," *Bell System Technical Journal*, 40(3):669–676, May 1961.

Kopec, G. E., *The Representation of Discrete-Time Signals and Systems in Programs*, PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1980.

Kopec, G. E., "The Signal Representation Language SRL," *IEEE Trans. Acoustics, Speech, and Signal Processing*, ASSP-33(4):921–932, August 1985.

Lam, M. S. L., *A Systolic Array Optimizing Compiler*, PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1987.

The Mathlab Group, *MACSYMA Reference Manual*, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1983.

Markel, J. D., "FFT Pruning," *IEEE Trans. Audio and Electroacoustics*, AU-19(4):305–311, December 1971.

Morris, L., "Automatic Generation of Time-Efficient Digital Signal Processing Software," *IEEE Trans. Acoustics, Speech, and Signal Processing*, ASSP-25(1):74–79, February 1977.

Myers, C. S., *Signal Representation for Symbolic and Numerical Processing*, PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1986.

Nagy, P. A. J., *MaMiS: A Programming Environment for Numeric / Symbolic Data Processing*, Master's thesis, Linköping University, Linköping, Sweden, 1988.

Oppenheim, A. V., and Schafer, R. W., *Discrete-Time Signal Processing*, Prentice-Hall, Inc., Englewood Cliffs, NY, 1989.

Prasanna, G. N. S., *Structure Driven Scheduling of Linear Algebra and Digital Signal Processing Problems*, PhD proposal, Massachusetts Institute of Technology, Research Laboratory of Electronics, Cambridge, MA, 1988.

Rader, C. M., "Speech Compression Simulation Compiler," *Journal of the Acoustical Society of America*, 37(6):1199, June 1965.

Rader, C. M., "A Simple Method for Sampling In-phase and Quadrature Components," *IEEE Trans. Aerospace and Electronic Systems*, AES-20(6):821–824, November 1984.

Rand, R. H., "Computer Algebra in Applied Mathematics: an Introduction to MACSYMA," Volume 94 of *Research Notes in Mathematics*, Pittman Advanced Publishing Program, Aulander, NC, 1984.

Regalia, P. A., and Mitra, S. K., "Kroenecker Products, Unitary Matrices, and Signal Processing Applications," *SIAM Review*, to be published 1989.

Richardson, J. M., *A Code-Division Multiple Beam Sonar Imaging System*, Engineer's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1989.

Sacks, E., *Qualitative Mathematical Reasoning*. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1982.

Schwartz, D. A., *Synchronous Multiprocessor Realizations of Shift-Invariant Flow Graphs*, PhD thesis, Georgia Institute of Technology, Atlanta, GA, 1985.

Signal Technology, Inc, *ILS: Interactive Laboratory System*, Goleta, CA 93117.

Singleton, R. C., "An Algorithm for Computing the Mixed Radix Fast Fourier Transform," *IEEE Trans. Audio and Electroacoustics*, AU-17(2):93–103, June 1969.

Siskind, J. M., Southard, J. R., and Crouch, K. W., "Generating Custom High Performance VLSI Designs from Succinct Algorithmic Descriptions," in *Proceedings, Conference on Advanced Research in VLSI*, pp. 28–40, 1982.

Skinner, D. P., "Pruning the Decimation in Time FFT Algorithm," *IEEE Trans. Acoustics, Speech, and Signal Processing*, ASSP-34(2):305–311, April 1976.

Smith, S. G., "Full Span Compilation of DSP Hardware," In *Proceedings ICASSP '87*, pp. 13.6.1–13.6.4, 1987.

Stefik, M. and Bobrow, D. G., "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, 6(4):40–62, Winter 1986.

Symbolics, Inc., *Symbolics Common Lisp: Language Concepts*, Symbolics, Inc., Cambridge, MA, 1986.

Traub, K. R., *A Compiler for the MIT Tagged-Token Dataflow Architecture*, L.C.S. Technical Report 370, Massachusetts Institute of Technology, Cambridge, MA, 1986.

Zissman, M. A., *An Array Computer for Digital Signal Processing*, Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1986.

Zissman, M. A., O'Leary, G. C., and Johnson, D. H., "A Block Diagram Compiler for a Digital Signal Processing MIMD Computer," In *Proceedings ICASSP '87*, pp. 43.1.1–43.1.4, 1987.