A Data Abstraction Language for Concurrent Programming

Thomas Peter Murtagh

Ph.D Thesis

TR 83-540
January 1983

Department of Computer Science
Cornell University
Ithaca, New York 14853

A DATA ABSTRACTION LANGUAGE FOR CONCURRENT PROGRAMMING

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Thomas Peter Murtagh

January 1983

Biographical Sketch

Thomas Peter Murtagh was born on March 28, 1953 in New York City. He graduated from Fordham Preparatory School in June 1970. He received the Bachelor of Arts degree in Mathematics from Princeton University in June 1974, and the Master of Science Degree in Computer Science from Cornell University in May 1976. He is a member of the Association for Computer Machinery and the I.E.E.E. Computer Society.

To my father

Acknowledgements

The encouragement of my advisor, Greg Andrews, made it possible for me to complete this thesis, and his careful guidance certainly improved its quality. I especially wish to thank him for having the patience to continue to advise my work by mail and over the phone after we had both left Cornell. I would also like to thank the other members of the Cornell Computer Science department who assisted me, particularly Alan Demers, Fred Schneider, and Peter Vanderbilt.

Finally, I especially thank my wife Fern, and my daughter Lindsey (age 3 months), who have had to spend too many evenings with me in the office or apart from me.

Table of Contents

1. Introduction 1

2. Relationships Between Synchronization and Data Abstraction Constructs 6

2.1 The Monitor and the Cluster 7

2.1.1 A Synchronized Buffer 9

2.1.2 An Unsynchronized Buffer 14

2.2 The Incompatibility of the Monitor and the Cluster 19

2.2.1 Relational Types 20

2.2.2 Synchronization 22

2.3 The Need for Unification 25

3. The Envi-0 Language 31

3.1 An Overview of Envi-0 32

3.1.1 The Specification Expression 33

3.1.2 Operations 36

3.1.3 Type Checking 37

3.1.3.1 Type Checking Rules in Envi 38

3.1.3.2 Type Specifications in Envi 42

3.2 Notation 48

3.3 Expressions 50

3.3.1 The Specification Expression 50

3.3.1.1 Component Specifications 51

3.3.1.1.1 Constant Declarations 52

3.3.1.1.2 Variable Declarations 54

3.3.1.1.3 Operation Declarations 56

3.3.1.1.4 Process Specifications 59

3.3.1.2 Type Checking Mechanisms 62

3.3.1.2.1 Local Names 65

3.3.1.2.2 Closure Rules 68

3.3.2 Identification Expressions 77

3.3.2.1 Local Names as Expressions 77

3.3.2.2 Component Selection 78

3.3.3 Invocation Expressions 82

3.4 Boolean and other Predefined Objects 88

3.5 Statements 91

3.5.1 Skip 91

3.5.2 Assignment 91

3.5.3 Invocation 94

3.5.4 Guarded Commands 96

3.5.5 The Input Statement 97

3.5.6 The Return Statement 101

3.6 An Example 102

3.7 Summary 107

4. Extending Envi-0 110

4.1 Abbreviations in Envi 111

4.1.1 Expression Syntax 111

4.1.2 Type Specifications 114

4.1.2.1 Source Restrictions 114

4.1.2.2 Content Restrictions 116

4.1.2.3 A Limited Macro Facility 118

4.1.3 Using Operations 120

4.1.3.1 Exclusive Operations 121

4.1.3.2 Procedures 123

4.1.4 Encapsulation 129

4.2 Built-in Types 134

4.2.1 Integers 135

4.2.2 Enumeration Types 137

4.2.2.1 The Type Character 137

4.2.2.2 Other Enumeration Types 140

4.2.3 Derived Types 141

4.2.3.1 Subrange Types 143

4.2.3.2 Arrays 148

4.3 Summary 153

5. Programming in Envi 155

5.1 Synchronization in Envi 156

5.1.1 The Readers/Writers Problem 158

5.1.2 A Readers Preference Solution 161

5.1.3 Writer's Preference 164

5.1.4 Alternating Readers and Writers 166

5.2 Simple Types and Polymorphism 168

5.2.1 The Rational Numbers 169

5.2.2 A Polymorphic Procedure 173

5.3 Queues 176

5.3.1 A Mergeable FIFO Queue 176

5.3.2 Priority Queues 185

5.3.2.1 The Queue Definition 185

5.3.2.2 An Alarm Clock 190

5.3.2.3 A Disk Scheduler 194
5.3.3 A Spooling System 201
6. An Examination of the Features of Envi 216
6.1 The Environ 217
6.1.1 Classes 218
6.1.2 Generalized Records 222
6.1.3 Synchronization Proposals 228
6.1.4 Data Abstraction Proposals 233
6.2 Expressions 240
6.2.1 Object Creation 240
6.2.1.1 Object Creation and Typing 241
6.2.1.2 Object Creation and Invocation 245
6.2.2 Expressions as Type Specifications 258
6.3 Summary 268
7. Conclusions 270
7.1 Summary 270
7.2 Further Research 272
Appendix A. Summary of ENVI-0 Syntax 277
Appendix B. Summary of Type System Closure Rules 279
Bibliography 280

Chapter 1
Introduction

In recent years, programming language designers in several areas of specialization have proposed modularization constructs that allow the programmer to combine and control access to logically related entities. Researchers concerned with the problems of concurrent programming have developed the Monitor [31], the Modula Module [67], the Resource [4], the Ada Task [57], and several similar constructs. Features such as the Form [58,69], the Cluster [45,46], the Capsule [17], and the Euclid Module [41] have been included in languages concerned with the definition of abstract data types. In addition, the proposals of Feldman [21] and Liskov [47,48] in the area of distributed computing call for similar constructs.

While these constructs are all based on common principles of modular programming and abstraction, they differ from one another in many ways. For example, the constructs proposed in connection with parallel programming languages use special rules for procedure invocation to provide synchronization, while data abstraction proposals provide no synchronization features. Also, most data abstraction constructs allow the programmer to include types as components,

hide the details of these types from code outside the construct and define binary operations on elements of the type, while synchronization proposals generally do not. In fact, a close examination will reveal many differences between synchronization constructs and data abstraction constructs in their mechanisms for component exportation, object creation, and type checking.

The differences between these constructs have resulted from efforts to specialize the general notion of a module to distinct problem areas. It is clear that the problems of concurrent programming are different from those of abstract type definition. The differences between a construct like the Monitor and one like the Cluster are just reflections of differences between these problem areas.

While it is obvious that these different problem areas have led to distinct constructs, it is not obvious that they _require_ distinct constructs. The fact that independent studies of constructs for different problems have led to different solutions does not imply that a common solution can not be found. In fact, the similarities between the constructs proposed for these distinct problems suggests that a common solution might exist. Nevertheless, several proposals have already been made for languages incorporating two similar but distinct modular constructs.

The most obvious such proposal is the Ada Programming language [57]. In Ada, two constructs, the TASK and the PACKAGE, provide the programmer with the ability to define objects as collections of simpler components. The Task is provided for synchronization. Its procedure-like components, entries, are based on a message-passing semantics. The Package is intended for data abstraction and separate compilation. It allows the programmer to include types as components and to hide their details from code outside of the construct. Liskov's guardian proposal [47,48] calls for a similar duplication of modular programming constructs.

The failure to unify the modular programming constructs in a language like Ada has two serious consequences. First, it complicates the syntax and semantics of the language without providing a matching increase in the language's capabilities. Secondly, it reduces the programmer's ability to produce programs that can be easily modified to meet changing needs. For example, in Ada, it is non-trivial to convert a Package that implements a queue into a synchronized implementation of a bounded buffer. Even though the code that implements the operations on these abstract types is similar, the introduction of synchronization constraints requires the use of a distinct set of language primitives.

This thesis presents a language, Envi, designed to correct these problems. The language has three major

features: a modular programming construct called the <u>specification expression</u>; a synchronization mechanism based on Andrews' operation construct [4], and a new form of type checking system. The specification expression provides the ability to describe and create abstract objects to be used in the solution of both synchronization and data abstraction problems. The objects produced by specification expressions are called <u>environs</u>. They are collections of components including constants, variables, and operations. The operation is a procedure-like construct for inter-process communication and synchronization. Each invocation of an operation transmits a message to the process that implements the operation. The implementing process can respond to invocations immediately or leave them pending. This enables the programmer to enforce synchronization constraints as needed. Envi's type system is based on the use of expressions as type specifications. It provides the means to encapsulate the definition of an object, as well as the ability to specify type restrictions that determine where objects can be used.

The language is unusual in that the mechanisms for object creation, synchronization and encapsulation are provided through these three separate but related features, rather than being combined in one complicated construct. For example, the specification expressions does not provide an "export list" or any similar feature to provide for encapsulation as do many other modular programming constructs. Instead, a definition is encapsulated by using the features of the type system together with those of the specification expression. This separation of mechanisms leads to a simpler and more powerful language. The language is simpler because it does not contain nearly as many different constructs. It is more powerful because the flexibility to combine its constructs in many ways provides capabilities not provided by languages containing distinct constructs for synchronization and data abstraction.

The second chapter of this thesis examines existing modularization constructs. In it, many of the statements made in this introduction are expanded and substantiated. The actual description of our proposal can be found in Chapters 3 and 4. Chapter 3 presents a minimal version of the language, stripped of all syntactic amenities; this exposes the nature of the underlying features. In Chapter 4, these features are supplemented by a more typical set of syntactic supports to illustrate the ways in which our proposals might be incorporated into more conventional languages. In Chapter 5, the features of the language are illustrated through numerous examples. Then in Chapter 6, these features are compared to those of related proposals. Finally, in Chapter 7, discusses implementation issues and other areas of further investigation suggested by this work.

Chapter 2
Relationships Between Synchronization and Data Abstraction Constructs

The main goal of this chapter is to completely describe the problem of supporting data abstraction and synchronization without introducing distinct modular programming constructs. It provides a more thorough explanation of this problem than that provided in Chapter 1. In addition, by examining the features and capabilities of other proposals, it substantiates our claims that these problems are important but remain unsolved.

A second goal of this chapter is to establish a framework in which our work can be discussed. The areas of language design, data abstraction and synchronization are filled with subtle ideas and vague terms to describe them. This can be a great obstacle to communication in these areas. Accordingly, as we discuss other language proposals, we will attempt to clarify the meanings of some terms that are used in this thesis.

These goals can be accomplished without examining each member of the large assortment of languages incorporating modular programming constructs. Although each language is unique, their similarities make it possible to cover the important issues by considering one representative of each of the major classes of modular constructs that are of interest. Therefore, our discussion is limited to one proposal for concurrent programming, the Monitor [31], and one proposal for data abstraction, the Cluster [46]. A more complete discussion of work related to ours is given in Chapter 6.

The material discussed in this chapter is divided into three major sections. First, we present brief introductions to the Monitor and the Cluster. Then, the incompatibilities between them are examined. This supports our claim that each of the approaches these proposals represent is too limited to replace the other. Finally, the third section presents the motivation for designing a construct combining the capabilities of proposals like the Monitor and the Cluster.

2.1. The Monitor and the Cluster

The Monitor and the Cluster are not only typical of modular programming constructs, they are also among the best known. The Monitor has been studied in detail [5,39,42,51] and included in several languages [8,22,67]. The Cluster has also been described in several publications [45,46]. This section does not attempt to provide complete descriptions of these proposals. A reader seeking such

descriptions is directed to the references given above. This section is concerned only with those aspects of these proposals most fundamental to our enquiry.

The discussion of the Monitor and the Cluster is organized around the solution of a simple programming problem. The problem is that of implementing a single element data buffer. This is just a very special case of the problem of defining a queue. The motivation for choosing a queue definition problem is that the problem is meaningful and interesting in both sequential and concurrent programming environments. The motivation for choosing such a trivial subcase of the queue definition problem was that it illustrates the most important features of the proposals without requiring complex code.

The specifications of the problem are quite simple. The buffer objects should be capable of storing a single value of some type, "item". Two operations, "insert" and "remove", are provided for manipulating the buffer. The "insert" operation takes a value of type "item" and places it in a buffer, if the buffer is empty. The "remove" operation is a function that returns the value stored in a buffer, when or if it is full, and leaves the buffer empty. In the sequential case, the implementation must recognize insertion into a full buffer as an overflow error. Such checking will be included in the Cluster solution. In the parallel case, insertion into a full buffer becomes a synchronization condition, requiring the inserter (producer) to wait for a remover (consumer). Similar differences appear in the implementation of "remove".

2.1.1. A Synchronized Buffer

An implementation of a single element buffer type using the Monitor construct is shown in Fig. 2.1. This definition, like that of all monitors, consists of two parts: the representation specification and the interpreting algorithms. The representation specification consists of those parts of the monitor that describe the variables used to represent the state of the abstract object. In the example, the representation specification consists of the declarations for "data", "full", "empty", and "nonempty". Type definitions in any language will contain this sort of information, because it is necessary to indicate which existing types will be used to represent elements of the new type. These types are called representation types.

In addition, a type definition must contain code that determines how the representation types will be used. We call this code the interpreting algorithms. In the monitor, this consists of procedures that may be referred to as components of the objects being defined, and initialization code executed when an instance of the object is created. The buffer type definition contains one line of


```
CLASS buffer:
    MONITOR
       BEGIN
          data : item;
          full : BOOLEAN;
          nonempty, empty : CONDITION;

          PROCEDURE insert( x : item );

               BEGIN

                    IF full THEN empty.wait;

                    data := x;
                    full := TRUE;

                    nonempty.signal

               END insert;


          PROCEDURE remove( RESULT x : item );

               BEGIN

                    IF NOT full THEN nonempty.wait;

                    x := data;
                    full := FALSE;

                    empty.signal

               END remove;

           full := FALSE;

       END buffer
```

Figure 2.1. A Monitor Definition of a Buffer.

initialization code,

```
full := FALSE
```

and two procedures, "insert" and "remove".

The name "buffer" defined in Fig. 2.1 can be used to create buffer instances through declarations such as

```
box : buffer
```

Whenever the scope containing such a declaration is entered, an instance of the single element buffer is created and the initialization code is executed. This new buffer can then be used by referencing the procedures 'insert' and 'remove' as components of 'box'.

Many of the details of the buffer objects created in this way can be understood without further explanation. They are only concerned with the sequential aspects of the buffer's behavior and the language features used are based on constructs commonly found in conventional languages. For example, the statements,

```
data := x
```

and

```
full := TRUE
```

have the same meaning in the "buffer" monitor as they would if used in a sequential programming language. Other

statements, including

```
IF full THEN nonfull.wait
```

and

```
nonempty.signal
```

are explicitly concerned with the problems introduced by parallelism. These statements depend on special features of the monitor proposal.

The condition variable is one such feature. A condition variable is a process queue. A running process can suspend itself and add itself to such a queue by executing the "wait" operation of a condition variable. If processes are waiting on the queue, a running process can suspend itself and activate one of the waiting processes by executing the "signal" operation of the condition variable. Executing "signal" has no effect when no processes are waiting.

In the example, "empty" and "nonempty" are condition variables. By executing the statement

```
IF full THEN empty.wait
```

in "insert", a process adds itself to the "empty" queue if the buffer is full. By executing

```
empty.signal
```

in "remove" a process which has emptied the buffer allows one of the processes in the queue to proceed. Thus, "empty" is the queue of processes waiting for the buffer to become empty. Similarly, "nonempty" is the queue of processes waiting for the buffer to be full so that they can perform removes.

Condition variables are only one part of the Monitor proposal's provisions for correct synchronization. While condition variables enable the programmer to enforce certain synchronization requirements explicitly, a more important synchronization constraint is enforced implicitly on all processes in a Monitor based system. No more than one process at a time is allowed to be executing within any monitor. If two processes attempt to invoke procedures of the same monitor simultaneously, one must be delayed until no other process is executing in the Monitor.

The effectiveness of this constraint depends on a third component of the Monitor proposal's mechanisms for synchronization. This is the limitation that the names of variables declared within the body of the monitor cannot be referenced by code outside of this body. This ensures that the restriction on parallel procedure calls implies that at most one active process can reference the variables of a monitor at any time.

These three features -- the prevention of parallel activity in monitor procedures, condition variables, and the restriction against external references to monitor variables -- are typical of the features found in all modular synchronization constructs. In order to prevent processes from interfering with one another by referencing variables simultaneously, all such constructs must either prevent parallel activity or provide the means to control it. In addition, they must enable the programmer to express synchronization constraints that depend on the state of the object being defined. Finally, they must statically enforce some degree of encapsulation to prevent code outside the module from circumventing the restrictions imposed by the first two features.

2.1.2. An Unsynchronized Buffer

A definition of a single element buffer type using the facilities of the CLU language [46] is shown in Fig. 2.2. One difference between this definition and the Monitor version is the absence of synchronization code. Since CLU does not provide features for parallel activity, errors rather than delays are produced when an insert is made into a full buffer or a remove is made from an empty buffer.

This difference, however, is not as significant as those that can be seen by examining CLU's approach to representation specification. In the Monitor version, the

```
buffer =
    CLUSTER IS create, insert, remove

        REP = RECORD [ data : item,
                       full : BOOLEAN ]

        create = PROC () RETURNS ( CVT );
                    RETURN( REP $ ( NIL, FALSE } );
        END create;

        insert = PROC( q : CVT, x : item );
                    IF full THEN  "error" ;
                    q.data := x;
                    q.full := TRUE;
        END insert;

        remove = PROC ( q : CVT ) : RETURNS( item );
                    IF NOT full THEN "error";
                    q.full := FALSE;
                    RETURN( q.data );
        END remove

END buffer
```

Figure 2.2. A CLU Definition of a Buffer.

state of a buffer is represented by the values stored in the variables "full" and "data". These variables are components of the monitor itself. The same variables appear in the CLU version, but they are defined as components of the elements of the type REP, which is in turn a component of the Cluster.

This difference is very important. Logically, the monitor describes an object consisting of both variables and procedures. This object both represents a buffer and provides the operations used to manipulate it. On the other hand, the cluster describes an object consisting of a type, REP, and several procedures. The cluster itself cannot represent a buffer because it has no variables as components. Rather, the objects belonging to its type component represent buffers. Accordingly, this type component is given the special name REP, which stands for representation. The object described by the cluster merely provides procedures that can be applied to elements of the type REP in order to manipulate them as buffers.

In CLU, the values of the REP type of a cluster are actually referred to by three different names in different contexts. Outside of the body of the cluster, the name of the cluster itself is used. Within the cluster, the name REP is used, except in parameter and return value specifications where the third name, CVT, is used. If we view a type strictly as a set of values, these names are all equivalent. In programming languages however, a type name also provides information concerning the operations that can be applied to objects. In CLU, each of the names associated with the representation type provides different information about the operations applicable to elements of that type.

When an object's type is described by the name REP, all of the operations associated with the type constructor used in REP's definition can be applied to the object. Thus, in the example, the "data" and "full" components of values of type REP are visible.

When an object's type is described by a cluster's name, none of the operations associated with the cluster's REP type can be applied to it. In the example, the components "data" and "full" cannot be selected from an object of type "buffer" because of this restriction. Furthermore, the name REP is not known outside of the cluster's body. Thus, the details of the representation used are inaccessible outside of the cluster's body.

The connection needed between the two types "buffer" and "REP" is provided by the third name "CVT", which is short for the word "convert". When it is used where a formal parameter type specification is expected, it indicates that the actual parameter used must belong to the type bearing the cluster's name, but that these values are to be converted into values of type REP for use by the procedure. This conversion involves no real computation, since all objects in the type associated with the cluster's name are also objects of type REP. Similarly, when "CVT" is used to describe a function's return value, it indicates that the objects returned must be of type REP, but that they are to

be converted into values of the type bearing the cluster's name for use in the context of the function's caller.

Like a monitor's name, a cluster's name can be used as a type name in a variable declaration. Thus, the declaration

```
box : buffer
```

is valid. Unlike the monitor proposal, this declaration does not produce an instance of the type buffer and bind it to box. Instead, box is bound to a location that can be used to store pointers to instances of the type. Accordingly, some other mechanism is needed to produce instances of the type. The "create" operation in Fig. 2.2 provides this facility.

"Create" returns a new element of the type "buffer" when it is called. It does not directly produce this element, however. Instead, it produces an element of the type "REP" and then converts this object into an object of type "buffer". The production of an object of type "REP" is accomplished using whatever facilities are provided with the representation type chosen. In this case, the built-in record constructor operation is used (i.e. REP$(NIL,FALSE)). The object is then converted into a "buffer" by simply returning it. The use of the type specification CVT indicates that conversion must be performed when "create" returns a value.

The details of this scheme are unique, but the underlying principles are typical of many data abstraction proposals. The modular constructs seen in these proposals enable a user to define a type whose elements are used to represent the objects being described and a group of procedures that associate a new interpretation with the elements of this representation type. Encapsulation is accomplished by providing different views of the type to code inside and outside of the module.

2.2. The Incompatibility of the Monitor and the Cluster

The preceding section described several differences between the Monitor and the Cluster. It did not, however, establish the significance of these differences. It is conceivable that the two proposals provide equivalent capabilities, even though they use different approaches. This section shows that this is not the case. In fact, we show that each of the proposals provides capabilities that the other lacks. The Monitor provides facilities for process synchronization that cannot be provided within the framework used in CLU. On the other hand, CLU's provisions for type definition cannot be matched by a language using the Monitor approach.

2.2.1. Relational Types

The major weakness of the Monitor approach is its inability to describe an entire class of data types that we will call the relational types. The common characteristic of these types is that the most important operations applicable to their elements require two or more parameters of the type involved. This class includes many of the common built-in types, including the integer and real numbers, and user defined types such as the complex numbers. The relationships between elements of these types are much more important than the structure of individual elements. As a result, the definition of binary operations is essential to the description of these types.

The description of a binary operation on elements of a type requires the ability to access the representations of two elements of the type simultaneously. The goal of encapsulation, on the other hand, places limitations on the ability of procedures to reference the details of each element's representation. In the case of the Monitor, the mechanisms used to achieve encapsulation interfere with the ability to define binary operations.

The problem becomes obvious if the monitor's approach to encapsulation is examined. If the monitor is used to define a type, each element of the type is represented by a distinct instance of the object described by the monitor construct. At least logically, each of these contains both the variables that represent the object and copies of the procedures that reference these variables. The representation is encapsulated because the variable associated with each instance can only be accessed by the copies of the procedures associated with the same instance. A procedure that references the representation of two elements of the type cannot be written in this framework. If its definition is not in the monitor it will be unable to reference any element's representation. If it is defined in the monitor, then each instance of the procedure will be associated with an instance of the type and only capable of accessing the representation of that instance.

In CLU, on the other hand, it is quite easy to describe a relational type. To illustrate this, a CLU implementation for a familiar relational type, the complex numbers, is shown in Fig. 2.3. While the monitor's procedures have access to the variables used to represent exactly one object because they are local to the context in which the variables appear, a cluster's procedures have access to the type REP, used to represent all of the objects of the new type, because their definitions are local to the context in which the type's definition appears. This privileged access to the REP type allows a cluster's procedures to access the representations of any elements of the new type passed to them using the parameter specifier CVT. Thus, the function


```
complex =
   CLUSTER IS
       create, add, sub, mult, ...

       REP = RECORD[ rpart : REAL,
                     ipart : REAL ];

       create =
            PROC ( x: REAL; y: REAL ) RETURNS( CVT );
                     RETURN ( REP$( x, y ) );
            END create;

       add =
           PROC( x: CVT, y: CVT ) RETURNS( CVT ) ;
                     RETURN ( REP${ x.rpart + y.ipart,
                                    x.ipart + y.ipart } );
           END add;

                     .
                     .
                     .

   END complex
```

Figure 2.3. Part of a Definition of the Type Complex.

'add' in the example defines a binary operation with access to the representation details of both of its parameters.

2.2.2. Synchronization

It is quite obvious that the monitor provides better synchronization features than the Cluster, because the

Cluster does not provide any. The weakness of the Cluster and other, similar constructs in the area of synchronization, however, goes beyond the obvious omission of appropriate features. The logical separation of the elements of a user defined type from the interpreting procedures, which proved a strength for type abstraction, makes it difficult to add good synchronization facilities to such languages.

The problem arises because the most common synchronization problems -- readers/writers, disk scheduling, bounded message queues, and others -- all involve types which include few or no binary operations. For example, the readers/writers problem is an abstraction that isolates the synchronization aspects of the problem of defining a type of data files. The set of primitive operations for such files typically would include open, close, read, write, reposition, and test for end of file. All these operations involve only one file. The problems faced by the programmer in defining such a type are quite different from those encountered in the definition of relational types. The emphasis is on the existence of multiple concurrent users of each object, rather than on the existence of multiple objects. Accordingly, constructs that enable the programmer to largely ignore the details associated with the existence of multiple instances are more appropriate for the definition of such types. The Monitor is such a construct; the Cluster is not.

The lack of facilities for describing interactions between instances of a monitor type enables the monitor proposal to provide simple, effective features for synchronization. The rule that at most one process can be active at any time within any of the procedures associated with a monitor is an example of such a feature. It is certainly simple. Its enforcement only involves information about the state of one instance of the monitor. It is effective because the procedures associated with any instance can only access the variables associated with the same instance. If a monitor's procedures could obtain access to variables in several instance of the monitor, more complex exclusion rules would be needed to account for these interactions. This is exactly what would be required if synchronization features were added to CLU.

Suppose that we attempt to apply a rule similar to the monitor rule to the procedures of a cluster. Whereas, at least logically, there are independent copies of a monitor's procedures, there is only one copy of the procedures of a cluster. Accordingly, the most obvious adaptation of the monitor's rule would prohibit parallel use of cluster procedures even if they were applied to distinct elements of the type involved. This would prohibit much useful concurrency. It seems that there should be some rule that would allow more parallelism while preventing interference. If one attempts to state one, however, it becomes obvious that it will not be as simple as the monitor's. How will it avoid placing unnecessary restriction on operations such as 'add(a,b)' and 'add(b,c)' which should be allowed to execute in parallel? Will it have to treat 'concatenate(a,a)' as a special case to avoid some form of self blocking? The effort needed to find an appropriate rule and the extra complexity which would probably result would be justifiable if the difficulties that made them necessary were significant. But, as we have explained, they arise because the cluter's approach is inappropriate to most synchronization problems, not because the problems themselves raise these issues.

2.3. The Need for Unification

The preceding sections have examined two approaches to object definition and argued that they provide significantly different capabilities. In the next chapter, a new approach that promises to provide the advantages of both will be presented. Two facts still need to be established in order to justify the development of this new approach. First, we must argue that it is desirable to have both of the capabilities described in one programming language. Then we must explain why it is undesirable to do so by providing two distinct object definition constructs. After all, if the capabilities involved are as different as we have argued, perhaps it is appropriate to provide them through separate mechanisms.

The first point is fairly easy to establish. Many researchers have argued for and worked toward the development of high level languages supporting parallel programming [3,4,19,6,8,9,10,32,31,49,67]. The need for such languages, which arose out of interest in the problems of operating system construction, has grown substantially with the emergence of distributed systems and other multi-processor systems based on microprocessor technology. Such parallel languages require synchronization facilities like those provided by the Monitor.

Others have described the potential advantages of providing good data abstraction facilities in programming languages and designed such facilities [11,23,25,26,41,44,45,46,53,54,58,60,64]. Their arguments apply to parallel languages as well as to any others. Thus, many language designers would share the belief that languages providing facilities for both data abstraction and synchronization are desirable. The inclusion of such facilities in the largest recent language design effort, Ada [57], is evidence of the acceptance of this belief.

It is somewhat harder to justify the assumption that separate facilities for synchronization and data abstraction are undesirable. This opinion is not as widely held. Ada, after all, does provide separate constructs, the Task and the Package. Furthermore, as mentioned above, the explanation of the differences between the Monitor and Cluster given in the preceding section seems to support Ada's approach. The resolution of this apparent contradiction depends on the recognition of a simple but important similarity between the Monitor and the Cluster.

The Monitor and the Cluster both define new data objects. This fact is so obvious that most discussions of these constructs ignore it and focus on either the mechanisms for synchronization or those for data abstraction. The definition of new objects is the primary function of these constructs, however. One cannot impose synchronization constraints or other limitations on references to an object that has not been defined and instantiated. Moreover, the Cluster and the Monitor use the same mechanism, aggregation, to describe and produce new object. By aggregation, we mean the process of joining existing objects together as components of a new object. This has been an important mechanism in programming languages for many years. The structures of PL/I [33] and COBOL [1] and the records of PASCAL [66] and its descendants are examples of aggregate data objects with variables as components. Many recent languages, following the lead of Simula 67 [16], have generalized this mechanism by allowing aggregates whose components include procedures and types. The Monitor, the Cluster and most other synchronization and data abstraction constructs are examples of this. Thus, while the forms of access

limitation associated with the Cluster and the Monitor are quite different, the objects being described are very similar.

In the process of designing special object definition constructs for synchronization and data abstraction, language designers have produced constructs that interlock three potentially independent language mechanisms. One is object description and creation. The second is type checking, or the static limitation of accesses to objects. The third is synchronization, or the dynamic limitation of accesses to objects. The Cluster combines a particular form of static access limitation and object creation. The Monitor combines particular forms of static and dynamic access limitation with object creation.

Language proposals that provide for both synchronization and data abstraction by defining two distinct constructs suffer from unnecessary complexity. As an example, consider the Task and Package of Ada. These constructs are both used to describe new objects. Ada provides a distinct object creation mechamism for each of these constructs. Task types are used to generate multiple instances of tasks, while generic definition facilities are used for packages. The inclusion of these two mechanisms obviously increases the languages complexity more than the inclusion of a single mechanism would have. More importantly, because the mechanisms perform such similar functions, the increased complexity is not accompanied by a comparable increase in programming power.

This approach to language design also limits a programmer's flexibility. There are objects which are meaningful in both sequential and parallel programming systems. For example, in the previous sections, a simple form of queue was used as an example. The description of the objects used to represent a queue and the static access constraints needed to encapsulate a queue's implementation are the same in both sequential and parallel environments. Therefore, one would hope that a queue definition made for a sequential program could be easily converted into one for a parallel program by adding synchronization specifications concerning dynamic access constraints. It would certainly seem unreasonable to have to rewrite the definition using a completely different construct. That, however, is exactly what would need to be done to convert a sequential queue implementation written using the Ada Package into a synchronized queue implementation in Ada. There are also objects which need to be synchronized but can be manipulated by some binary operations resembling those associated with the relational types discussed above. The use of separate constructs like the Task and Package make their description very difficult.

Thus, there is no contradiction in our opinions. Synchronization and data abstraction problems are significantly different, as argued in section 2. Their solutions, however, involve the use of common mechanisms. In the next chapters, a language is presented that shows that the overlapping constructs found in languages like Ada can be replaced by three orthogonal language features.

Chapter 3

The Envi-0 Language

In this and the next chapter, the details of the proposed synchronization and data abstraction constructs are presented. The description of constructs like these raises an irksome problem. They are best understood when incorporated in a complete language. The design of a complete language, however, is a much larger task than the design of the constructs themselves. The most common solution to this problem is to construct a language based on some currently popular language around the new constructs. The large number of PASCAL variants produced recently attests to the popularity of this approach. We will use this technique, but make one important variation in recognition of a danger inherent in it.

The danger to which we refer is that the features borrowed may hide those that are new. This can happen in several ways. Readers unfamiliar with the language that is copied are obviously likely to have problems separating the new work from the borrowed. Readers familiar with the copied language can also be misled. Unless radically different syntactic conventions are used for the new features, it is easy to assume that syntactic similarities indicate

semantic similarities when the truth is otherwise.

The variation we will make to reduce this danger is to define two languages incorporating our proposals. Both languages will be named after a new object description construct, the Environ. The first, Envi-0, includes a minimum of borrowed features. Furthermore, in this language, we have not hesitated to use unusual notations for new constructs. It is intended to present the essentials of the proposals with as few obscuring details as possible. It is, however, so sparse a language that it would be quite cumbersome to use. The second language, Envi, is an extension of Envi-0 that includes abbreviations for the most common uses of Envi-0's features and includes several built-in objects. It provides an example of how the essentials of our proposals could be used to form the basis of a full language.

Envi-0 is described in the remainder of this chapter; Envi in the next. Unless we are specifically discussing differences between the two languages, the name Envi will be used to refer to both.

3.1. An Overview of Envi-0

Before describing Envi-0 in detail, this section presents a brief, informal discussion of the most important and unusual features. The features that are discussed correspond exactly to the three mechanisms -- object description, dynamic access limitation and static access limitation -- discussed in the last chapter. They are the specification expression, which is used for object description; a special procedure mechanism called the operation, which provides actions on objects and can be used to enforce dynamic access limitations; and Envi-0's type checking system, which provides for static access limitations.

3.1.1. The Specification Expression

The *specification expression* is the Envi-0 construct used to describe and create data objects. It has the form:

```
ENVIRON identifier ( component-declarations )
```

Component declarations for constants, variables, operations and processes are allowed. As an example, the skeleton of a specification expression that might be used to describe a buffer similar to that discussed in Chapter 2 is shown in Fig 3.1. This specification expression contains declarations for two variables and one procedure that are to be components of the described objects. The notation

```
FROM[ item$create(...)]
```

and the similar specifications used elsewhere in the example are type specifications. The one above is equivalent to the type 'item' used in the Monitor example. The meaning of these specfications will be discussed in Section 3.1.3.


```
ENVIRON buffer
   ( data :   VAR( FROM[item$create(...)]);
     full :   VAR( FROM[boolean$create(...)]);
     insert : OP( FROM[item$create(...)]);
                    .
                    .
                    .
     )
```

Figure 3.1. The Form of A Specification Expression.

While the overall structure of this construct is not unusual, its role in the language is. As its name implies, it is an expression. Its evaluation produces an instance of the object it describes. Such an object is called an environ. When one describes a product to a salesman, one expects to get the product delivered, not the tools used to produce it. Similarly, when one describes an object in a programming language, one should expect to be given the object, not something that can be used to produce it later. Unfortunately, this is not the case in many languages. In these languages, constructs similar to the specification expression are used as type specifications. The elaboration of such constructs produce templates that can be used to produce instances of the objects described, rather than the objects themselves. For example, in Pascal, record definitions appearing in type declarations produce record types, even though they describe the records themselves. Thus, treating such constructs as expressions makes more sense. Furthermore, it has several practical advantages that will be discussed in Chapter 6.

A second unusual characteristic of the specification expression is the role that the objects it describes play in Envi-0's data space. In many languages, aggregate data objects form a class of data values that are treated differently from the members of the language's built-in types. In particular, the use of aggregates in assignments, in parameter passing and as the return values of functions is often limited. The restriction on the use of records as function return values in Pascal illustrates this. In Envi-0, this distinction is not made. Environs can be used in any context where a data value can be used. In fact, all data values in Envi-0 are environs. After all, even the built-in types of most languages are just aggregates of bits.

Finally, the specification expression is used not only to describe data objects used in a program, it is used to describe programs themselves. In Envi, a complete program will be described as a single environ.

3.1.2. Operations

Envi incorporates a message-passing based system for procedural abstraction and synchronization proposed by Andrews [4]. This system provides a construct called the operation which combines properties of conventional procedures with message passing.

In Envi, an operation can be declared as a component of an environ. Such an environ must also contain a process that "implements" the operation. The operation component can be invoked by other processes as if it were a procedure. When an operation is invoked, the actual parameters are sent to the implementing process and the calling process waits for a reply.

An implementing process responds to invocations of operations by executing a construct called the input statement. This construct resembles Dijkstra's guarded command statements [20], except that each guarded command is replaced by an input command. An input command contains the name of an operation from which a message can be received and a statement list to be used to process this message. Thus, the statement

```
IN f(x) --> S1
[] g(y) --> S2
NI
```

would indicate that an invocation of operation f or g is expected. If an invocation of 'f' occurs, S1 will be executed with 'x' bound to the message value received. Similarly, if an invocation of 'g' occurs, S2 will be executed with 'y' bound to the message value received. If both operations are invoked simultaneously, one of the invocations will be selected non-deterministically and processed. After execution of S1 or S2, the process which invoked the operation and the process which received the request both resume their execution independently.

The main differences between Envi-0's use of operations and Andrews' proposal are simplification made in areas where the details of Andrews' proposal either went beyond the scope of our interest or were made unnecessary by other features of Envi.

3.1.3. Type Checking

The most unusual features of Envi-0 involve type checking. For example, the variable declaration for 'data' shown in the 'buffer' specification expression of Fig. 3.1 used

```
FROM[ item$create(...)]
```

where the monitor and cluster simply used 'item'. This strange notation is a consequence of the decision to separate the mechanism for object creation from the mechanism for static access control. In a language that provides

a special construct for the definition of types, such as the CLU cluster, it is easy to use the names associated with instances of the construct as type names. In Envi-0, as we will see, no special construct is used to describe the objects that represent types; they are described using the general object description mechanism: the specification expression. This makes it more difficult to use object names as type names, but does not mean that Envi-0 cannot describe such types. Envi-0 provides a system of type specification designed around the assumption that all objects, including types, are environs.

This section introduces the principles behind this type system. The discussion is divided into two subsections. The first compares the form of Envi's type checking rules to those of other languages. The second compares the use of Envi's type specifications to those of other languages

3.1.3.1. Type Checking Rules in Envi

Conventional type checking systems can be decomposed into three kinds of rules. The first associates a type description with each expression in a program. The second determines the type required by each context where an expression appears. Finally, the third states how to compare the type associated with an expression to that associated with the context in which it is used. The diagram in Fig. 3.2 illustrates the role of these three parts of a type checking system.

As a simple example, consider the Pascal statement

```
a := b
```

in a program where 'a' has been declared as an integer variable and 'b' has been declared using the subrange type

```
1..100
```

The declaration of 'b' causes Pascal to associate the type

```
1..100
```

with the expression 'b'. The declaration of 'a' and the rule for assignment statements associate the type 'integer' with the context in which 'b' appears. Finally, the rule for comparing types determines that '1..100' can be used

```
expression                 context
    |                         |
    |                         |
    |                         |
    V                         V
type of e    ---- ? --->  type required
```

Figure 3.2. The Structure of Most Type Checking Systems.

where 'integer' is expected.

Type checking in Envi-0 has a different but related structure. There are no rules for associating types with expressions, because there are no types in the sense of a conventional language. The absence of a special object description construct for user defined types necessitated their elimination. There are rules corresponding to the second and third components of a conventional type system, but they do not associate types with contexts and compare types. Doing so would be useless since types are not associated with expressions. Instead, they associate expressions with contexts in the program, and describe how to compare an expression used in a context with the expression associated with the context. Thus, the structure of Envi-0's type system is described by the diagram in Fig. 3.3.

```
                                   context
                                      |
                                      |
                                      |
                                      V
expression used ---- ? ----> acceptable expression
```

Figure 3.3. The Structure of Envi-0's Type Checking System.

Although replacing conventional type specifications with expressions may seem unreasonable, there are two observations involving the role of expressions in programming languages that support this approach. First, even in conventional languages, type specifications and expressions serve a similar function. They both describe sets of values. It is obvious that a type specification does this. It is normally its only purpose. Expressions, on the other hand, are generally thought of as describing values, not sets of values. Many expressions, however, produce different values each time they are evaluated. Therefore, if an expression is to be viewed statically, before execution, it must be viewed as a description of the set composed of all the values it may produce during execution.

The second observation is that it is possible to speak of comparing expressions in the same way that the third component of a conventional type system compares types. Others have previously suggested basing type checking on equivalence relations over expressions [56]. In fact, the rules for comparing types in a conventional type checking system induce an ordering on expressions. If the symbol '$\geq$' is used to denote this ordering, and 'a' and 'b' are expressions, then '$a \geq b$' holds if 'a' can be used in any context where 'b' can be used according to the rules for type checking. Given such an ordering on expressions, type constraints can be expressed without reference to type names by

associating a 'minimal' expression with each context where an expression is needed. If 'b' is the minimal expression for a context, then 'a' can be used in the context if '$a \geq b$'.

Envi-0 uses a very similar ordering of expressions to define type correctness. This ordering is defined in terms of the syntax of a program by rules presented later in this chapter. If '$a \geq b$' holds for two expressions we will say that 'a' dominates 'b'. In addition, rules are given that associate minimal expressions with every context where expressions are needed in a program.

3.1.3.2. Type Specifications in Envi

In the declaration of procedures, programmers normally supply type specifications for formal parameters; these are later compared to the types of actual parameter expressions to determine the correctness of calls. In Envi-0, a programmer must supply a description of the minimal expression acceptable as an actual parameter. This is done by placing the expression in square brackets preceded by the keyword 'FROM'. Such a construct is called a type restriction. Similar restrictions are used to describe return values and data values in variable declarations. Thus, the program segment

```
FROM[ item$create( ... )]
```

mentioned in the first paragraph of this section is an example of a type restriction.

The use of ellipses in the restriction

```
FROM[ item$create( ... )]
```

requires some explanation. These ellipses are part of Envi-0's syntax. The syntax for expressions in Envi-0 is extended by allowing the actual parameter expressions in a function call to be replaced by ellipses. Obviously, such an expression cannot be evaluated. In most languages, that would make it useless. In Envi-0, however, expressions have a role other than the production of values. They are also used to describe sets of values. An expression of the form

```
f(...)
```

is interpreted as a description of the set of all values that could be produced by valid calls of the function 'f'. Such an expression proves to be very valuable for specifying type constraints.

At this point, the reader has probably realized that type restrictions in Envi-0 are not going to be easy to understand at first. This is true, but not because they are inherently complicated. Actually, Envi-0's type checking system involves fewer details than a conventional system of comparable power, such as Alphard's [69]. This is because

by eliminating type specifications the need for a type specification notation and a set of rules for associating type specifications with expressions has been eliminated. It seems likely, therefore, that the real reason that Envi-0 type restrictions appear complicated is that the unusual syntax makes it hard to associate them with ideas about types that the reader already possesses. Accordingly, before concluding this overview, we will consider how some common type constraints are expressed in Envi-0.

Type specifications employ two common approaches to the description of sets of objects. In the first, a set of objects is specified by describing a set of components that all the objects share. We call this approach <u>restriction by content</u>. In the second, a set of objects is described by specifying some common element used in their production. We call this approach <u>restriction by source</u>. The difference between structural equivalence and name equivalence [65,61] illustrates these two approaches. In structural equivalence, objects are considered to be of the same type if they possess exactly the same components. In name equivalence, objects are of the same type only if they are produced through the same type description. Structural equivalence involves the use of restriction by content because it is based on the components of the objects involved. Name equivalence involves the use of restriction by source because it depends on the ways in which the objects are produced. As will be shown, however, structural equivalence and name equivalence are just special cases of these two general restriction mechanisms.

To understand the use of restriction by content in Envi-0, consider the facilities used to place restrictions on the type parameters to generic definitions in languages like CLU and Alphard. Fig. 3.4 shows the header used for such a definition in CLU. The full definition would associated the name 'sorted_bag' with a type generator that could be applied to an existing type to produce a new type. Restrictions are placed on the types to which 'sorted_bag' can be applied by the 'WHERE' clause in the definition. In particular, it requires that all such types provide binary

```
sorted_bag =
   CLUSTER[ t : TYPE ] is create, insert, ...
        WHERE t HAS
              lt, eq : PROCTYPE( t,t ) RETURNS(bool);
                            .
                            .
                            .
    END
```

Figure 3.4. The Header of a Generic Cluster Definition.

operations named 'lt' (less than) and 'eq' (equal). If a type is viewed as an aggregate consisting of the operations that can be applied to the elements of the type, then this is an example of restriction by content. It is different from structural equivalence, however. Structural equivalence requires that objects possess *exactly* a certain set of components, while the where clause requires objects to possess *at least* a certain set of components. To distinguish these two forms of restriction by content, we will call the second *component covering*. It is the method supported by Envi-0.

Restrictions based on component covering are expressed in Envi-0 by using specification expressions in type restrictions. Suppose one wishes to require that only environs with at least the components 'lt' and 'eq' be used as parameters to a certain operation. A restriction of the form shown in Fig. 3.5 would be used. This restriction is correct because only expressions that produce objects with the desired components can be used in any context in which the expression included in the type restriction is valid. Any specification expression can be used in this way to specify a type constraint based on restriction by content. In general, a restriction of the form:

```
FROM[ ENVIRON ( D ) ]
```

where D is a list of declarations describes the set of all

```
FROM[
 ENVIRON t
  (lt:OP(x,y:FROM[t$create(...)]): FROM[boolean$create(...)];
   eq:OP(x,y:FROM[t$create(...)]): FROM[boolean$create(...)]
    )
   ]
```

Figure 3.5. A Restrictor Requiring 'lt' and 'equal'.

objects possessing at least the components described in D.

The other major approach to type restriction, restriction by source, can also be used in Envi-0. Such restrictions are expressed by using function calls in type restrictions. For example, the restriction

```
FROM[  integer$create( ... ) ]
```

which contains a call of the 'create' function of the environ named 'integer' is used to describe the type of integers. The logic behind this is probably best explained by again examining typing in CLU. The use of a cluster name as a type specification in CLU is an example of restriction by source. The normal understanding of such a type specification views the module described by the cluster as the source of all objects of the type. A closer examination

reveals that the true source of objects of such a type is the collection of all procedures using CVT as a return value specification declared in the cluster. In Envi-0, we simply narrow our view by requiring that a single operation act as the source of all elements of a type. This is not a severe restriction, since most user type definitions in languages like CLU already possess a function, typically called 'create', that serves this purpose. Once such an operation is identified, an expression consisting of a call of the function can be used to form a type restriction describing the type.

3.2. Notation

In the following sections the details of Envi-0 are presented. The syntax of the language is described by a grammar expressed in a modified version of BNF similar to that used in [69]. Each rule of the grammar consists of a left hand side of one non-terminal symbol and a right hand side composed of one or more syntactic entities. Several rules involving the same left hand side can be abbreviated by including all of the right hand sides in one rule separated by the alternation symbol, '|'. Syntactic entities include non-terminals, terminal strings, delimiters and any of the abbreviations described below. Non-terminals are formed by enclosing a suitable name for the strings they generate in angle brackets ('<' and '>'). Thus,

< expression > and < id > are possible non-terminals. Delimiters include the characters ',', ';' and '[]'. The following abbreviations are allowed.

1) Following any syntactic entity other than a delimiter a superscript '*' indicates that the entity may be repeated zero or more times. A superscript '*' following a delimiter indicates that the entity preceding the delimiter may be repeated zero or more times with the delimiter used as a separator between instances. Thus,

a b* a

describes the strings

a a ; a b a ; a b b a ; a b b b a ; . . .

while

a b,* a

describes the strings

a a ; a b a ; a b , b a ; a b , b , b a ; . . .

2) Following any syntactic entity other than a delimiter, a superscript '+' indicates one or more occurrences of the entity. If the '+' follows a delimiter, one or more instances of the entity preceding the delimiter are allowed with the instances separated by the delimiter. That is, '+' behaves like '*' except that entities modified by '+' may not be completely omitted.

3) Several syntactic entities may be grouped to form a single entity by surrounding the group with braces ('{' and

'}'). Such groupings are used to apply a repetition specification to a string of symbols. Thus,

{ < id > : < type > };*

describes a list of zero or more identifier-type pairs separated by semi-colons. Simply surrounding an entity or group of entities with braces indicates that they are optional. For example,

{ PROCESS < stmt list > END }

describes an optional process specification.

A summary of all of the rules in the Envi-0 grammar is included as Appendix A.

3.3. Expressions

Envi-0 provides three types of expressions: specification expressions, identification expressions and invocation expressions. Specification expressions were discussed in the language overview. The other forms are more closely related to expressions used in conventional languages. Identification expressions correspond roughly to variables and constants. Invocation expressions are essentially function calls.

3.3.1. The Specification Expression

A specification expression is a list of component specifications separated by semicolons, surrounded by parentheses, and preceded by the keyword 'ENVIRON' and an optional identifier. This is summarized by the rule

< expr > ::= ENVIRON (< id >) (< comp spec >;*)

The result of evaluating such an expression is an object containing the components described in the expression's body. Such an object is called an <u>environ</u>. The values produced by all expressions in Envi are environs. If the optional identifier is included in a specification expression's header, it can be used within the component specifications to refer to the instance of the object with which the components are associated. This identifier's scope is limited to the body of the expression.

3.3.1.1. Component Specifications

Specification expressions can contain four types of components: constants, variables, operations and processes. The specifications for constants, variables and operations differ from those for processes in that they associate names with the objects they describe. In recognition of this, the specifications of constants, variables and operations are called declarations. All component specifications appear in specification expressions. Syntactically, these facts are reflected in the rule


```
< comp spec > ::=  < const decl >
                |  < var decl >
                |  < op decl >
                |  < process >
```

3.3.1.1.1. Constant Declarations

The simplest form of declaration used in Envi-0 makes a <u>permanent</u> association between a name and an environ and makes this environ a subpart of the environ produced by the specification expression in which the declaration occurs. Such declarations are called <u>constant declarations</u>. They are syntactically described by the rule

< const decl > ::= < id >,$^+$: < expr >

That is, a constant declaration is formed by following a list of identifiers by a colon and an expression. During the evaluation of a specification expression containing such a declaration, the expr provided is evaluated once for each name in the identifier list. Each of the objects so produced is bound to one of the names in the identifier list and all of them are included as components of the object produced by evaluating the specification expression.

Examples of possible constant declarations include

totalsize : integer$10

and

```
table : tree$create( integer$100 )
```

In the first, the right hand side of the declaration is an identification expression which names an object that is to be bound to the component name 'totalsize'. The names used have been chosen to suggest that such a declaration might be used to bind a name to the object representing the integer 10. As will be seen, integer is not a built-in type in Envi-0. This and several other examples in this chapter have been written under the assumption that a user defined implementation of the integers has been programmed and bound to the name 'integer'. References to integer constants and operations will therefore be made by selecting components of this object using the selection operator, '$'. The definition of such an object is presented in Chapter 4.

The right hand side of the second example is an invocation expression or function call. The effect of the declaration is to bind the result of this call to the name 'table'. Here, the names have been chosen to suggest that 'table' is being bound to an instance of a tree that holds up to 100 elements.

The provision for name lists allows the programmer to abbreviate the declaration of several constant components in a declaration of the form


```
table1, table2 : tree$create( integer$100 )
```

Because the expression used in a constant declaration is evaluated once for each component name declared, this declaration would cause two invocations of the 'tree$create' operation and bind 'table1' and 'table2' to the distinct instances of the tree type produced. (If only one evaluation were performed, 'table1' and 'table2' would be bound to the same object.)

3.3.1.1.2. Variable Declarations

The second form of declaration associates a name with a variable which can later be bound to values through assignment statements. The syntax of such declarations is described by the rules

```
< var decl > ::=
      < id >,+ : VAR( < type rest > (INIT < expr > } )
< type rest > ::= FROM[ < expr > ]
```

That is, a variable declaration consists of a list of names followed by a colon and a variable description. A variable description consists of a type restriction and an optional initial value expression. During the evaluation of a specification expression containing such a declaration, a distinct variable is associated with each name listed in the declaration. All of these variables are included as components of the environ produced by the specification expression containing the declaration. If an initial value expression is included in a variable description, it is evaluated once for each name listed in the declaration and the objects produced are bound to the variables created.

A very simple example of a variable declaration would be

```
temp: VAR( FROM[ integer$create( ... ) ] )
```

As suggested in the overview of the language, the type restriction

```
FROM[ integer$create( ... ) ]
```

describes the set of all objects used to represent integers. Thus, the declaration associates 'temp' with an integer variable.

Several variables of the same type can be described by listing all of their names before the variable description as in

```
low, ptr, high : VAR( FROM[ integer$create( ... ) ] )
```

Initialization of variables can be accomplished by including an expression that will produce the desired value as in

```
size : VAR( FROM[ integer$create( ... )] INIT integer$0 )
```

The initialization expression used in such a declaration

must satisfy the type restriction given. In terms of the ordering on expressions, this implies that the initialization expression must dominate the expression used in the type restriction.

3.3.1.1.3. Operation Declarations

Operations are the means through which processes interact in Envi-0. Within the language, their use resembles that of procedures and functions, but they can also be viewed as message channels. In procedure definition constructs, the code which processes call and the specification of the mechanisms used for parameter transmission form parts of a single construct. In Envi-0, an operation declaration is only concerned with the transmission of parameters (messages). The code is specified separately in the process that implements the operation. As a result, an operation declaration looks like a procedure header without a procedure body.

The exact syntax of operation declarations is described by the rules

```
< op decl > ::= < id >,+ : < op desc >
< op desc > ::= OP( < parm spec >;* ) [: < type rest > ]
< op desc > ::= OP( ... ) : < type rest >
< parm spec > ::= ( < id >,+ : ] < type rest >
```

That is, an operation declaration consists of a list of identifiers followed by a description of the parameters that can be passed through the operation. This description normally consists of a list of parameter specifications, which describe the type restrictions placed on values included as parameters in invocations of the operation, and an optional type restriction, which describes the values that may be returned to the invoker of the operation. Thus,

```
modulo : OP( d, n: FROM[integer$create(...)]
                   ) : FROM[integer$create(...)]
```

```
insert : OP( x     : FROM[integer$create(...) ];
             table : FROM[    tree$create(...) ] )
```

and

```
create: OP( ... ) : FROM[integer$create(...)]
```

are possible examples of operation declarations.

If the declaration appears in a specification expression which is itself part of a type restriction, the parameter specifications may be omitted. Such a declaration is distinguished from the declaration of an operation that takes no parameters by replacing the parameter specifications by ellipses. This allows the programmer to construct type restrictions that require an object to have a certain operation without describing its parameterization. Thus, the declaration given for 'create' above might be used in a type restriction to require that an object possess a

'create' operation whose parameterization is unknown. Such declarations are often used to describe operations which are never called but are used in other type restrictions. Examples will be given later.

As a result of the separation between the code that specifies the implementation of an operation and the declaration that describes its parameterization, the identifiers included in parameter specifications do not function as conventional formal parameter names. They are not used to refer to the actual parameter values by the code that implements the operation. The names used for this purpose are introduced separately by the input statement, as described in Section 3.5.5. The scope rules of the language, which are presented in Section 3.3.1.2.1, limit the scope of names introduced by parameter specifications to the operation declaration in which they appear. As a result, in most operation declarations, they serve no purpose and the rules of Envi-0 syntax allow them to be omitted. Thus, the declaration of 'insert' above could be written as

```
insert : OP( FROM[integer$create(...)];
             FROM[ tree$create(...) ] )
```

The use of these names, however, can clarify the intended use of the parameter.

There are operation declarations, however, in which it is essential to include names in parameter specifications. In Envi-0 it is possible to define polymorphic operations by using one or more of the names defined in parameter specifications in the type restrictions given for other parameters or the return value. Thus, if the type restriction 'FROM[s]' describes a type definition,

```
join: OP( t: FROM[ s ]; x,y: FROM[t$create(...)] )
```

might be used to declare an operation which takes a type 't' as its first parameter and two elements 'x' and 'y' of type 't' as additional parameters. In such a declaration, the parameter name 't' is essential.

3.3.1.1.4. Process Specifications

Processes are the fourth type of component that can be included in an environ. A process specification consists of a list of statements surrounded by the brackets 'PROCESS' and 'END'.* Thus, the syntax for process specifications is summarized by the simple rule

```
< process > ::= PROCESS < stmt >;+ END
```

The evaluation of a specification expression containing such

* Envi processes have no local variable declarations. These were omitted because they complicated the description of the type rules and were not needed in any of the examples considered.

a process specification causes the creation of a new process which executes the statements. The process terminates when all of the statements have been executed.

This description of the syntax of process specifications is deceptively simple because it ignores the details associated with the statement list included in the specification. Envi-0 provides three simple statement types: the null statement, the assignment and a call statement. These simple statement types can be combined in statement lists or in conditional and iterative constructs based on the guarded command proposal of Dijkstra [20]. Finally, Envi-0 provides a construct called the input statement similar to that proposed by Andrews [4]. This construct allows the programmer to instruct a process to wait for an invocation of one of several operations and to indicate how such an invocation should be processed when it occurs. Within an input statement, a return statement may be used to return values to the invoker of an operation. A complete description of these statement types is given in Section 3.5.

Process components of environs are unnamed. They can only be referred to indirectly through the operations they implement. The relationship between processes, operations and the other component types is illustrated by the simple example shown in Fig. 3.6. This specification expression describes an object which acts like the 'tickers' used by

```
ENVIRON ticker
  ( click : OP() : FROM[ integer$create(...) ];
    count : VAR( FROM[ integer$create(...) ] );

    PROCESS
       ticker$count := integer$0;
       DO boolean$true -->
          ticker$count := integer$succ( ticker$count );

          IN click () --> RETURN( ticker$count ) NI
       OD
    END
    )
```

Figure 3.6. A Call Counter.

gate keepers to count crowds. It contains three components. This first is an operation named 'click'. This operation takes no parameters and returns an integer value. Each time it is called it returns a count of the total number of times that it has been called. This value is kept in the second component of the environ, an integer variable named 'count'. Finally, the third component is a process that implements the operation. First, the process initializes the variable 'count' to zero. Then, it enters an infinite loop that repeatedly causes it to wait for an invocation of 'click' and execute an assignment and a return statement whenever one occurs. Again, the name integer is being used as one would expect a user defined implementation of the integer

type to function in Envi-0. In particular, we assume that its component 'succ' implements the successor function. The Boolean type is also used. This type is built-in to the Envi-0 languge and is discussed in Section 3.4.

3.3.1.2. Type Checking Mechanisms

Type checking in Envi-0 is based on an ordering defined on the expressions in a program. There are two sorts of rules involved in the definition of this type system. The first defines the ordering itself. In the overview, the ordering was explained by stating that '$a \geq b$' should imply that 'a' can be used in any context where 'b' could be used. Here, precise rules that define an ordering with this property in terms of the syntax of a program are given. The second form of rule explains how to apply the ordering to determine the type correctness of a given construct.

The rules that define the ordering on expressions are themselves divided into two groups. The first associates a finite relation with each distinct scope; it will be called the *minimal relation* for the scope. The second states closure properties which must be true of the ordering used for type checking. The ordering actually used for type checking in a given scope is that obtained by applying the *closure rules* to the minimal relation for the scope.

A few observations might clarify the distinction between these two groups of rules. It is obvious from the intuitive explanation of the ordering given that distinct orderings must be associated with each scope in a program. For example, within the specification expression shown in Fig. 3.7 the name 'a' can be used to refer to the object described by the specification expression. Therefore, the type ordering should reflect the fact that 'a' can be used in any context where the entire expression would be valid. That is, the relationship shown in Fig. 3.8 must hold in the ordering. On the other hand, outside of the body of the specification expression the name 'a' may be undefined or may be bound to some other object. It would therefore be incorrect if the preceding relationship held in the orderings associated with all parts of the program. Instead, distinct orderings must be associated with each scope. The

```
ENVIRON a
   ( x : VAR( FROM[ integer$create(...) ] );
     y : VAR( FROM[ integer$create(...) ] )
               .
               .
               .
     )
```

Figure 3.7. A Simple Specification Expression


```
a ≥ ENVIRON a
       ( x : VAR( FROM[ integer$create(...) ] );
         y : VAR( FROM[ integer$create(...) ] )
                    .
                    .
                    .
         )
```

Figure 3.8. Minimal Relation Information About 'a'.

minimal relations associated with each scope capture these differences.

At the same time, there are many aspects of the ordering that should be fixed in all scopes. For example, regardless of the meaning of 'a', the relationship shown in Fig. 3.9 should be true because the first expression includes all

```
ENVIRON a                                  ENVIRON a
 ( x: VAR( FROM[ t ] );       ≥          ( x: VAR( FROM[ t ] )
   y: VAR( FROM[ t ] )                         )
  )
```

Figure 3.9. Relating Two Specification Expressions.

the components of the second. The closure rules account for these considerations.

3.3.1.2.1. Local Names

The main function of the minimal relation associated with a scope is to ensure that the meanings of names like 'a' in Fig. 7 are correctly reflected in the expression ordering. These names are called local names. They differ from component names in that they can be used without qualification within their scopes of definition. Two ways of introducing local names have already been described. The first is obviously to include a name in the header of a specification expression. The second is to include identifiers in the parameter specifications used in operation declarations.

Local names in Envi are subject to Algol-like scope rules. The scope of a local name introduced in the header of a specification expression is the expression itself. Within the expression, the name refers to the object described by the expression. At the same time, any information about other identifiers declared outside the body of the expression should remain true within the body.

These scope rules allow a single identifier to take on different meanings in different parts of a program. While this is convenient for the programmer, it is not essential.

Obviously, any program can be transformed into a program in which all local names are unique by suitably substituting unused names for multiple occurrences of other names. Accordingly, any rules defining the semantics of programs can be simplified by restricting them to programs in which all local names are unique without any loss of generality. This approach is taken in the description of the type checking rules of Envi.

The first of these rules defines the minimal relation associated with each scope in a program. The minimal relation associated with the body of an expression of the form

```
ENVIRON a ( D )
```

is formed by adding

$$a \geq \text{ENVIRON } a \text{ (} D \text{)}$$

to the minimal relation associated with the scope in which the expression appears.

The rule for handling the introduction of local names through parameter specifications is similar. If an operation declaration contains a parameter specification of the form

```
a : FROM[ x ]
```

the name 'a' can be used to refer to the actual parameter values throughout the operation declaration. As will be seen in the discussion of invocations, the type restriction

FROM[x]

ensures that all actual parameter expressions used in this parameter position will dominate 'x'. Accordingly, the type system should assume that

$a \geq x$

in determining the validity of uses of 'a'. Therefore, the rule for associating a minimal relation with the scope of an operation declaration containing parameter specifications of the form

a_i : FROM[x_i]

is to add

$a_i \geq x_i$

to the minimal relation associated with the specification expression in which the operation declaration appears for each such parameter specification. Parameter specifications of the form

b_1, ..., b_n : FROM[x]

should be considered as shorthand for the list of specifications

b_1 : FROM[x];
.
.
.
b_n : FROM[x];

in applying this and any other rule of the type system.

One other construct, the input statement, that introduces local names will be introduced later. The preceding cases, however, illustrate the nature of the minimal relation. For each scope in the program it gives a list of all of the names that can be used without qualification and indicates how each one can be used by associating it with an expression that is less than or equal to it.

3.3.1.2.2. Closure Rules

The minimal relation provides all the basic facts about a program needed to do type checking. The closure rules provide the information about the language needed to relate these facts. All of the closure rules for the expressions of Envi-0 are presented in this and the following sections. A summary of these rules is included in Appendix B.

The first closure rules are the rules of transitivity and reflexivity.

Rule TR. Transitivity

If a, b and c are expressions and

$a \geq b$ and $b \geq c$

then

$a \geq c$

Rule RF. Reflexivity

If a is an expression, then

$a \geq a$

It should not be surprising that these rule are included, since we have referred to the relation that we are defining as an ordering, but it is worth observing that they are consistent with the intuitive explanation given for the ordering. Reflexivity is obvious. For transitivity, note that if 'a' is valid in any context where 'b' is valid and 'b' is valid in any context where 'c' is valid, then it is certainly reasonable to assume that 'a' is valid in any context where 'c' is valid.

The third rule is one of several rules that formalize the idea of component covering discussed in Section 3.1.3.2. These are called structural similarity rules for the specification expression. The simplest of these rules expresses the idea that if one specification expression has all of the component declarations that a second possesses, then it should be usable in any context where the second is usable.

Rule SS1. Component Deletion

If 'b' is obtained by deleting one or more component specifications from the specification expression 'a', then

$a \geq b$

To see how these rules work, recall the 'ticker' example used above and repeated in Fig. 3.10. While the complete type checking rules of Envi-0 have not yet been discussed, the fact that a component selection of the form

a$b

is only type correct if

```
ENVIRON ticker
  ( click : OP() : FROM[ integer$create(...) ];
    count : VAR( FROM[ integer$create(...) ] );

    PROCESS
      ticker$count := integer$0;
      DO boolean$true -->
         ticker$count := integer$succ( ticker$count );

         IN click () --> RETURN( ticker$count ) NI
      OD
    END
    )
```

Figure 3.10. A Call Counter.

a ≥ ENVIRON (b : FROM[x])

or

a ≥ ENVIRON (b : VAR(FROM[x]))

should be clear. In fact, this will be given as the rule for determining the correctness of such expressions in Section 3.3.2. Given only this rule and the minimal relation, the expression 'ticker$count' used in 'ticker' would be considered incorrect, since neither

ticker ≥ ENVIRON (count : FROM[x])

nor

ticker ≥ ENVIRON (count : VAR(FROM[x]))

are true in the minimal relation. However, the minimal relation includes the information shown in Fig 3.11. and SS1 together with transitivity therefore implies that

ticker ≥ ENVIRON(count: VAR(FROM[integer$create(...)]))

Thus, these two rules formalize a simple form of component covering.

The remainder of the closure rules for specification expressions handle various examples where component covering occurs in a form not recognized by SS1. For example, simply reodering the component specifications in a specification


```
ticker ≥ ENVIRON ticker
          (click: OP() : FROM[integer$create(...)];
           count: VAR( FROM[integer$create(...)] );

           PROCESS
            ticker$count := integer$0;
            DO boolean$true -->
              ticker$count:=integer$succ(ticker$count);

              IN click() -->  RETURN( ticker$count ) NI
            OD
           END
            )
```

__Figure 3.11__. __Information from the Minimal Relation__.

expression should not affect its structure for the purpose of type checking. This leads to the rule

__Rule SS2__. Component Reordering

If 'a' is obtained by reordering the component specifications of a specification expression 'b', then

b ≥ a   and   a ≥ b

The rules presented thus far do not recognize that two specification expressions with identical component declarations but different local names are identical. This is corrected by the rule

Rule SS3. Environ Renaming

If 'b' is obtained from the specification expression 'a' by replacing all occurrences of the local name used in 'b' by a new name that does not introduce any name conflicts, or by adding a local name that does not introduce any name conflicts, then

$a \geq b$ and $b \geq a$

The remaining structural similarity rules deal with particular component types, rather than with specification expressions in general. The first applies to constants. For the purposes of type checking, it is not necessary to require that two constant components have the same value for one to cover the other. It is merely required that the 'type' of one be 'compatible' with the other. In Envi, this notion of compatibility is made precise by the rule:

Rule SS4. Constant Compatablity

If 'a' is a specification expression with a constant component

```
c : x
```

for some expression 'x', and 'b' is obtained from 'a' by replacing the declaration of 'c' by

```
c : y
```

where 'y' is an expression such that

$x \geq y$

then

$a \geq b$

There are two rules for variable components. The first is similar to that just given for constants. It recognizes the fact that one variable component should cover another if the type of the second contains all elements in the type of the first.

Rule SS5. Variable Type Compatability

If 'a' is a specification expression with a variable component declaration of the form

```
c : VAR( FROM[ x ] )
```

for some expression 'x', and 'b' is obtained from 'a' by replacing the declaration of 'c' by

```
c : VAR( FROM[ y ] )
```

where 'y' is an expression such that

$x \geq y$

then

$a \geq b$

The second accounts for the fact that the type system should ignore initialization expressions when comparing two specification expressions.

Rule SS6. Variable Initialization Hiding

If 'a' is a specification expression with a variable component declaration of the form

```
c : VAR( FROM[ x ] )
```

for some expression 'x', and 'b' is obtained from 'a' by replacing the declaration of 'c' by

```
c : VAR( FROM[ x ] INIT y )
```

where 'y' is an expression such that

$y \geq x$

then

$a \geq b$ and $b \geq a$

The requirement that 'y' dominate 'x' in this rule ensures that the component declaration formed using 'y' will be type correct.

Three structural similarity rules are provided for operation declarations. The first provides for differences in the return value type specifications used in otherwise compatible components. This is similar in effect to the first rule for variable components.

<u>Rule SS7</u>. Operation Type Compatability

If 'a' is a specification expression containing a declaration of the form

c : OP(π) : FROM[x]

for some parameter specifications 'π' and some expression 'x', and 'b' is obtained from 'a' by replacing the declaration of 'c' by

c : OP(π) : FROM[y]

where 'y' is an expression such that

$x \geq y$

then

$a \geq b$

The second allows for local name substitutions. It recog-

nizes the fact that if the only differences between two operation declarations are the names introduced in the parameter specifications, then they describe the same operation.

<u>Rule SS8</u>. Operation Parameter Renaming

If 'a' is a specification expression with a declaration of the form

c : OP(π) : FROM[x]

for some list of parameter specifications 'π' and some expression 'x', and 'b' is obtained from 'a' by replacing all occurrences of some local parameter name introduced in 'π' by some other name or by introducing a local parameter name where one had been omitted without introducing any name conflicts, then

$a \geq b$ and $b \geq a$

Finally, the third provides the ability to 'forget' an operation's parameterization.

<u>Rule SS9</u>. Operation Parameterization Hiding

If 'a' is a specfication expression containing a declaration of the form

c : OP(π) : FROM[x]

and no names introduced in 'π' are used in 'x', then if 'b' is obtained from 'a' by replacing the declaration of 'c' by

c : OP(...) : FROM[x]

then

$a \geq b$

As explained in the section on operation declarations, the ability to describe an operation without describing its parameterization is useful in conjunction with restriction by source. Rule SS9 enables the type system to handle such descriptions.

3.3.2. Identification Expressions

In Envi-0, an identification expression is one that describes an object by giving a name to which it is bound. Two forms of identification expressions can occur in Envi-0: local names and component selections.

3.3.2.1. Local Names as Expressions

The simplest form of expression in Envi-0 is a local name used to describe the object to which it is bound. The syntax of such expressions is simply

< expr > ::= < id >

Although there are three construct which introduce local names in Envi, only local names introduced in the headers of specification expressions and in input guards can be evaluated. Local names introduced in operation declarations can only be used in expressions that occur within the type restrictions used in the operation declaration. Therefore, they are never evaluated.

The result of evaluating a local name is simply the environ it names. If the name was introduced in the header of a specification expression, it refers to the instance of the object described by the specification expression. If the name was introduced in an input guard it refers to one of the actual parameters included in the invocation being processed; the details of the evaluation of these names will be discussed when the input statement is described below.

3.3.2.2. Component Selection

Expressions formed by selecting a constant or variable component of an object are also classified as identification expressions in Envi-0. Syntactically, such an expression consists of a sub-expression that describes an environ, an identifier that names a component of the environ, and a '$' to connect the two. The sub-expression is called a qualifying expression.

The rule

< expr > ::= < expr > $ < id >

summarizes this syntax. The expression

ticker$count

seen in the example used above is an instance of component selection in which the qualifying expression is the local name 'ticker'. Evaluation of such selections proceeds form left to right. It is also possible to use invocations and specification expressions as qualifying expressions. Thus,

table$search(value)$x

might select the 'x' component from the value returned by the function 'search'.

The closure rules that have been presented do not directly determine type correctness. Instead, they define a relation that is used by the type checking rules to determine type correctness. The first of these rules is for component selections.

If 'a' and 'd' are expressions and 'b' and 'c' are identifiers, then

a$c

is a valid expressions only if

a $\geq$ ENVIRON b (c : d)

or

a $\geq$ ENVIRON b (c : VAR(FROM[d]))

In addition to this type checking rule, there are two closure rules associated with component selections. The first is another structural similarity rule.

<u>Rule SS10</u>. Selection Similarity

If 'a' and 'b' are expressions such that

a $\geq$ b

then if 'b$c' is a valid expression, 'a$c' is a valid expression and

a$c $\geq$ b$c

This rule says that if 'a' is usable in any context where 'b' is usable, then any selection involving 'a' should be usable wherever the corresponding selection involving 'b' is usable.

The second component selection rule is the first closure rule other than transitivity not based solely on structural similarities. Instead, it attempts to relate expressions with different structures through the declarations given in the program.

<u>Rule CS</u>. Component Selection Resolution

If 'a' and 'b' are expressions and 'c' and 'y' are identifiers such that

a $\geq$ ENVIRON y (c : b)

or

a $\geq$ ENVIRON y (c : VAR(FROM[b]))

and 'x' is the expression obtained from 'b' by replacing all occurrences of 'y' in 'b' by the body of 'a', then

a$c $\geq$ x

This rule essentially says that if 'a' has a constant or variable component 'c' whose declaration indicates that it is bound to a value that can be used in any context where 'b' can be used, then 'a$c' should be usable in any context where 'b' can be used. This would normally be expressed by saying that

    a$c ≥ b

but the rule introduces a new expression 'x' and concludes that

    a$c ≥ x

The reason for this is simple. The expression 'b' appears inside a specification expression that may form a scope distinct from that in which 'a$c' appears. Within the scope where 'b' appears, 'y' denotes the object to which the component 'b' belongs. In the identification expression, on the other hand, 'a' denotes this object. Accordingly, to produce an expression that is equivalent to 'b' in the scope where 'a$x' appears, we replace any occurrences of 'y' by 'a'.

The function of the CS rule can be seen by returning to the expression

    ticker$count

in the 'ticker' example. Previously, rule SS1 was used to conclude that

    ticker ≥ ENVIRON(count:VAR(FROM[integer$create(...)]))

This establishes the fact that the expression is type

correct. Together with the rule CS and the rule of transitivity, it also implies that

    ticker$count ≥ integer$create(...)

This means that 'ticker$count' can be used in any context where any call of the form 'integer$create(...)' can be used. In more conventional terms, 'ticker$create' is recognized as an expression with type 'integer' by the type system. Thus, rule CS enables the system to extract the type information provided in constant and variable declarations.

3.3.3. Invocation Expressions

If an environ possesses an operation component whose declaration includes a return value specification, then an invocation of this operation can be used as an expression. Such expressions are called <u>invocation expressions</u>. They have three parts. First, a qualifying expression that describes the environ possessing the desired operation is needed. Then, the name of the operation component must be given. Finally, a possibly empty list of actual parameter expressions is included. If the expression appears as part of a type restriction, the actual parameter expressions can be replaced by ellipses. Syntactically, these parts are combined according to the rules

    < expr > ::= < expr > $ < id > (< expr >;*)

```
< expr > ::= < expr > $ < id > ( ... )
```

For example, the expression

```
table$search( value )
```

uses the local name 'table' as a qualifying expression, the identifier 'search' as an operation name, and the local name 'value' as an actual parameter expression.

The evaluation of an invocation expression begins with the evaluation of all the sub-expressions included. The environ produced by evaluating the qualifying expression will contain an operation component identified by the name used in the invocation expression. The values produced by the parameter expressions will be sent as a message to this operation. The process evaluating the invocation expression will then wait until a message is received in response. The value in this message will be used as the value of the invocation expression. The mechanism through which this return value is actually produced will be discussed with the input statement in Section 3.5.5.

The type system of Envi-0 includes rules for determining the validity of invocation expressions and closure rules that relate them to other expressions through the expression ordering. These rules are similar to those for identification expressions but they are somewhat more complicated because invocation expressions include more sub-expressions.

Accordingly, to avoid repetition of some of the details, the rule for determining the validity of an invocation expression is combined with a closure rule.

The validity of an invocation expression depends on two factors. First, the type system must determine that any value produced by the qualifying expression will have the operation component being invoked. In addition, each of the actual parameter expressions must be checked against the corresponding parameter specification in the operation's declaration. Rules IV1 and IV2 state these requirements formally. Rule IV2 handles the special case of expressions in which the parameter expressions have been omitted.

<u>Rule IV1</u>. Invocation Resolution 1

Given a group of expressions

$$a_0, a_1, \ldots, a_n, b_0, b_1, \ldots, b_n$$

and identifiers

$$f, x_0, x_1, \ldots, x_n$$

If expressions $c_0, \ldots, c_n$ are formed from the b_i's by replacing all occurrences of the identifiers $x_0, \ldots, x_n$ by the corresponding a_i, and

1) $a_0 \geq \text{ENVIRON } x_0 \; (\, f : \text{OP}(x_1 : \text{FROM}[b_1]; \ldots; x_n : \text{FROM}[b_n]) : \text{FROM}[b_0]\,)$

2) $a_i \geq c_i$

for each i in the range 1 to n,

and

3) all of the a_i's involved in the substitutions performed to form the expressions c_1 through c_n are constant expressions.

then

$a_0\$f(a_1; \ldots ; a_n)$

is a valid invocation expression. Furthermore

$a_0\$f(a_1; \ldots ; a_n) \geq c_0$

<u>Rule IV2</u>. Invocation Resolution 2

Given a group of expressions,

$a_0, a_1, \ldots, a_n, b_0$

and two identifier, f and x_0, if

$a_0 \geq$ ENVIRON x_0
(f : OP(...) : FROM[b_0])

and c_0 is obtained by replacing any occurrences of x_0 in b_0 by a_0, then

$a_0\$f(\ldots)$

is a valid invocation expression. Furthermore

$a_0\$f(\ldots) \geq c_0$

The first requirement of rule IV1, that a_0 dominate the specification expression given, ensures that a_0 is an expression that will produce only objects possessing the operation component being invoked. The requirement that $a_i \geq c_i$ ensures that each actual parameter expression is usable in any context where the b_i used in the corresponding parameter specification is valid. As in the rule for identification expressions, the c_i's are used instead of the b_i's

because it is necessary to make certain substitutions in order to obtain expressions that are logically equivalent to the b_i's but stated in terms of the identifiers accessible in the scope where the invocation expression appears.

The third requirement reflects the fact that certain limitations must be placed on the substitutions made to produce the c_i's. The local names used in the b_i's refer to the values produced by the particular evaluation of the expressions a_0 through a_n performed as part of the evaluation of the invocation expression. In general, the a_i's may produce different values with each evaluation. Allowing the substitution of such expressions in the type restrictions of other parameters could lead to insecurities in the type system. For example, if the operation declared by

```
join: OP( t: FROM[ s ]; x,y: FROM[t$create(...)] )
```

were called by the expression

```
mod$join$( tgen$type(k), u, v )
```

the expressions 'u' and 'v' will be compared to the expression

```
tgen$type(k)$create(...)
```

to determine the correctness of the invocation. This requirement does not ensure that they are both members of the type represented by the object produced by the evaluation of the expression

tgen$type(k)

used in the invocation, it only ensures that they belong to the type represented by an object that was produced by some evaluation of this expression. If each evaluation of this expression produces a different object, then the desired type restriction will not be enforced. Accordingly the third reqirement on invocations is that only constant expressions may be substituted for local parameter names used within parameter type restrictions. A constant expressions is defined to be either a local name or an expression formed by selecting a constant component from a constant expression.

Together, these three requirements ensure that the invocation is valid. The final statement that

$$a_0\$f(a_1; \ldots ; a_n) \geq c_0$$

is a closure rule for invocations. This rule states that the return value specification given in the operation's declaration determines the contexts in which an invocation of the expression can be used validly.

Finally, there are two structural similarity rules for invocation expressions.

<u>Rule SS11</u>. Invocation Similarity 1

If $a_0, \ldots a_n, b_0, \ldots b_n$ are expressions such that

$$b_i \geq a_i$$

for all i, and 'f' is an identifier such that

$$a_0\$f(a_1, \ldots, a_n)$$

is a valid invocation expression, then

$$b_0\$f(b_1, \ldots, b_n) \geq a_0\$f(a_1, \ldots, a_n)$$

<u>Rule SS12</u>. Invocation Similarity 2

If a_0 and b_0 are expressions such that

$$b_0 \geq a_0$$

and 'f' is an identifier such that

$$a_0\$f(\ldots)$$

is a valid invocation expression, then

$$b_0\$f(\ldots) \geq a_0\$f(\ldots)$$

3.4. <u>Boolean and other Predefined Objects</u>

Many useful but non-essential features have beem excluded from Envi-0 in order to keep the language simple. The collection of data types built into most languages is such a feature. In this languages the only type that is critical to the language's definition is the Boolean type. This is because of its importance in conditional and iterative constructs. This section describes the definition of the Boolean type in Envi-0.

The key to the introduction of Booleans or any other built-in objects to Envi-0 is the recognition of a detail not mentioned in the definition of minimal relations given above. The rules to associate a minimal relation with each scope describes the new relations in terms of the relations associated with the surrounding scopes. A program, however, must have an outermost scope. This implies that in order to complete the definition of the minimal relations, a standard minimal relation associated with the scope surrounding all programs must be specified. It is here that any built-in names are accounted for. Rather than being empty, this minimal relation will contain information about built-in names. The type Boolean provides us with an unavoidable example of the way in which this is done.

An implementation of the Boolean values must provide the operations 'and' and 'or', the unary negation operation and the constants 'true' and 'false'. It is impossible to describe the implementation of the operations 'and', 'or' and 'neg' within the Envi-0 language. The collection of Boolean operations and constants, however, may still be viewed as an environ, and its structure can be described precisely by writing a specification expression with the components just mentioned. If such an expression is written, one can provide the type system with all the information needed to check the type correctness of uses of 'boolean' by informing the type system that the name

'boolean' dominates this expression. This is done by stating that the relationship shown in Fig. 3.12 holds in the minimal relation associated with the scope surrounding all programs. The environ on the right of the '≥' describes the structure of the object that is logically associated with the name 'boolean'. This makes it unnecessary to provide special case rules in the type system to handle Boolean values.

To describe the type of Boolean values using Envi-0's type restrictions, we must assume the existence of a function that creates all Boolean values. In Envi-0, such an operation is used in type specifications to describe the set of all elements of the type. The 'create' component

```
boolean ≥
    ENVIRON boolean
     ( create: OP( ... ) : FROM[ ENVIRON( ) ];

      neg: OP( x: FROM[boolean$create(...)]
                 ):FROM[ boolean$create(...) ];

      or,and: OP( x,y: FROM[boolean$create(...)]
                     ):FROM[ boolean$create(...)];

      true : boolean$create(...);
      false: boolean$create(...)      )
```

Figure 3.12. Relationship Defining Boolean.

declared in the expression shown above is included for this purpose.

The return value restriction associated with the 'create' operation ensures that boolean values are treated as atomic objects in Envi-0 programs. This type restriction forces code outside of the 'boolean' environ to treat Boolean values as if they had been produced by the expression

```
ENVIRON()
```

Since this expression describes objects with no components, users of the Boolean type are prevented from attempting to select components from Boolean values. As a result, the only ways a program can manipulate Boolean values are through assignment and by passing them to the operations that are components of the object 'boolean'.

3.5. Statements

Statements request that some action be performed. They differ from expressions in that they do not produce values. This section describes the statement types of Envi-0 in detail.

3.5.1. Skip

A statement of the form


```
< stmt > ::= SKIP
```

has no effect.

3.5.2. Assignment

Assignment in Envi-0 is accomplished by selecting a variable component of some environ and specifying a new object that should be associated with this component. The syntax is fairly standard. It is expressed by the rules

```
< stmt > ::= < var > := < expr >

< var > ::= < expr > $ < id >
```

The only unusual aspect is that each variable specification must include a qualifying sub-expression whose evaluation will produce the environ whose component is to be modified.

The statement

```
ticker$count := integer$0
```

is an example of an assignment in Envi-0. The environ bound to the '0' component of the environ 'integer' is assigned to the 'count' component of 'ticker'. As in other contexts where components are selected from environs, expressions other than local names can be used. Thus,

```
table$search(value)$data := integer$0
```

assigns a new value to the 'data' component of the environ produced by a call to the function 'search'. The name 'integer' is used here to aid the user's intuition, although the type integer is not discussed until Chapter 4.

Assignment in Envi-0 associates a variable with a value without copying the value. That is, after the assignment

```
a$b := c
```

the expressions 'a$b' and 'c' denote the same object, not two copies of an object. If the object has components that can be changed, changing them through 'c' will affect the object described through 'a$b'. This form of assignment is unusual, but it has been used in other languages -- notably CLU and Simula 67. It is included in Envi because it provides a way to create multiple access paths to an object that is to be shared by several processes.

The final aspect of assignment is the rule determining the type correctness of an assignment statement. This rule must ensure that the objects produced by the qualifying expression are certain to have the variable component named by the identifier used in the variable description. Also, it must ensure that the expression on the right hand side will produce values that can be assigned to this variable. Both of these requirements are expressed by the following rule.

If 'a', 'b', and 'c' are expressions, 'x' and 'v' are identifiers, and 'd' is an expression obtained from 'c' by replacing all occurrences of 'x' by the body of 'a', then

```
a$v := b
```

is valid if

1) a ≥ ENVIRON x
   (v : VAR(FROM[c]))

2) Either 'x' does not occur in 'c' or 'a' is a constant expression, and

3) b ≥ d

Again, substitution is used in this rule to obtain a 'd' that approximates the meaning of 'c' in the scope of the assignment. As a result, the requirement that

b ≥ d

states that any expression assigned to a variable must be usable in any context where the expression used in the variable's type restriction could be used.

3.5.3. Invocation

The invocation statement is syntactically equivalent to the invocation expression. It is described by the rule

< stmt > ::= < expr > $ < id > (< expr >,*)

The subparts described on the right hand side of this rule play the same semantic roles as in the specification expression. Execution begins with evaluation of all sub-expressions. The value produced by the first expression must have an operation component named by the identifier used in the statement. The values produced by the remaining expressions are joined to form a message which is transmitted to the implementing process. The process executing the invocation statement waits until this message is received and processed. Unlike the invocation expression, however, no value is returned to the invoker. The mechanisms used by the implementing process are discussed in the description of the input statement below.

The rule for determining the type correctness of an invocation statement is also nearly identical to that for the invocation expression.

Given a group of expressions

$a_0, a_1, \ldots, a_n, b_1, \ldots, b_n$

and identifiers

$f, x_0, x_1, \ldots, x_n$

if an expression, c_i, is formed from each b_i by replacing all occurrences of the identifiers $x_0, \ldots, x_n$ by the corresponding a_i, then

$a_0\$f(a_1; \ldots ; a_n)$

is a valid invocation statement if

1) $a_0 \geq$ ENVIRON x_0 (f : OP(x_1: FROM[b_1]; ...; x_n: FROM[b_n])

2) $a_i \geq c_i$

hold for each i in the range 1 to n, and

3) all of the a_i's involved in the substitutions performed to form the expressions c_1 through c_n are constant expressions.

The only difference between this rule and that given above for invocation expressions is the omission of references to the return value.

3.5.4. Guarded Commands

Envi-0 uses the *if* and *do* statements of Dijkstra's guarded command language[20]. Statements of these types are composed of lists of guarded commands separated by boxes ([]) and surrounded by the brackets IF and FI or DO and OD. A guarded command is just an expression and a statement list separated by an arrow. The expression is called the guard. It must dominate, i.e. be "≥", the expression 'boolean$create(...)'. The statement part of a guarded command is only executed when the boolean value produced by evaluation of the associated guard is true. In the case of an IF statement, the entire statement is executed by selecting one command with a true guard and executing its statement part. On the other hand, a DO statement is executed by selecting one command with a true guard, executing the statement part associated with the guard, and then repeating the process until no command with a true guard can be found.

An example of an IF statement is

```
IF integer$lt( x, integer$0) --> x := integer$neg(x)
[] integer$ge( x, integer$0) --> x := x
FI
```

This statement sets x equal to its absolute value if one assumes that the integer operations used have their customary interpretations.

The syntax rules for these statements are

```
< stmt > ::= IF < guarded command > [] + FI

< stmt > ::= DO < guarded command > [] + OD
```

where

```
< guarded command > ::= < expr > ---> < stmt >;+
```

3.5.5. The Input Statement

An input statement enables a process to indicate that it is prepared to receive invocations of one or more operations and to specify how it will process them. The input statement is syntactically similar to the IF and DO statements. Each input statement consists of one or more input commands separated by boxes ([]) and bracketed by the keywords IN and NI. An input command contains the name of an operation, a list of formal parameter names, a statement list to be executed when an invocation is selected, and

optionally, a Boolean expression. Together, the operation name, parameter names and the Boolean expression form an input guard. This syntax is summarized by the rules

```
< stmt > ::= IN < input command > []+ NI

< input command > ::=
    < id >( < id >,* )(WHEN < expr >) ---> < stmt >;+
```

Execution of an input statement causes a process to wait until one of the input commands can be been executed. An input command can only be executed if an unreceived message is pending in the queue associated with the operation named by the command and the Boolean expression following WHEN, if one appears, evaluates to true. If more than one input command can be executed, one is chosen nondeterministically.

The execution of an input command proceeds in two steps. First, the actual parameter values passed in the message are bound to the names appearing as formal parameter names in the input guard. Then, the statement list given in the command is executed. During the execution of these statements, the use of a formal parameter name as an expression will produce the value of the corresponding actual parameter. When the execution of these statements is completed, the execution of the input command and the entire input statement also completes, and the process that invoked the chosen operation is allowed to proceed.

The fact that the formal parameter names used in an input guard can be used as expressions in the input command indicates that each input command forms a distinct scope. Rules for associating minimal relations that describe the proper use of the parameter names with these scopes must be provided.

An input command of the form:

$f(\ p_1,\ \ldots,\ p_1\ )\ \ldots$

is valid only if the operation 'f' is declared in the specification expression immediately containing the command syntactically and is not used in input commands within other processes in this expression. The operations declaration must be of the form:

$f\colon OP(\ p_1\colon FROM[e_1];\ \ldots;\ p_n\colon FROM[e_n]\ )$

or

$f\colon OP(\ p_1\colon FROM[e_1];\ \ldots;\ p_n\colon FROM[e_n]\ )\ \colon FROM[e_0]$

where the parameter names may be omitted as explained previously.

The intent of the parameter specifications in the declaration is that each p_i will be bound to an object usable in any context that a value produced by the

corresponding e_i would be usable. Thus, the minimal relation for the scope of the input command should be formed by adding a pair of the form

$p_i \geq e_i$

to the relation associated with the surrounding scope for each formal parameter name. This will allow the type checking system to recognize valid uses of these local names.

The WHEN clause allowed in the input statement is intended for the specification of synchronization constraints. For example, assuming the existence of an integer implementation as we have in other examples, the statement

```
IN P() WHEN integer$gt( sem$val, integer$0)
          ---> sem$val := integer$pred( sem$val )

[] V()    ---> sem$val := integer$succ( sem$val )
NI
```

implements the P and V operations on semaphores. The definition of the scope of formal parameter names, however, includes the WHEN clause. Accordingly, formal parameter names can also be used in WHEN clauses. The intent is that a message can only be received if evaluating the WHEN clause with any formal parameter names bound to the values it contains produces true. This can be used for synchronization or to express other constraints on the input values to an operation. For example, if 'div(x,y)' is an operation that divides 'x' and 'y', then

    IN div(x,y) WHEN y ≠ integer$0 ---> ...

would prevent division by zero by refusing to ever process a message requesting such a division.

3.5.6. The Return Statement

The return statement allows the code in an input command to specify a return value. The form of the return statement is:

    < stmt > ::= RETURN(< expr >)

Such a statement is executed by evaluating the expression and sending the value produced to the invoker of the operation. The execution of a return statement also terminates the execution of the input command in which it occurs. A return statement can only appear in the statement list of an input command associated with an operation whose declaration specifies that a return value will be produced, and a return statement must be executed in each such input command.

If a return statement appears in an input command, the declaration of the operation must be of the form:

$$f\text{: OP(} p_1\text{: FROM}[e_1]\text{; ...; } p_n\text{: FROM}[e_n] \text{) : FROM}[e_0]$$

for expressions e_0 through e_n and identifiers p_1 through p_n. The closure rule for invocation expressions assumes that an

operation that returns values will only return values satisfying the return value type restriction given in the operation's declaration. Therefore, a return statement of the form

    RETURN(x)

is only type correct if

$$x \geq e_0$$

3.6. An Example

In the last chapter, the single element buffer problem was used to illustrate the features of the Cluster and the Monitor. Now that the features of Envi-0 have been described, this example can be used again to show how the main features of our language interact. In this section, a solution to the single element buffer problem in Envi-0 will be shown. This solution closely parallels the monitor solution. Envi type definitions more similar to the CLU definition given above are presented in Chapters 4 and 5.

First, consider the simpler problem of describing an object that represents a buffer. A specification expression that describes an object similar to that used to represent buffers in the monitor solution is shown in Fig. 3.13. The declarations of the variables used in the monitor have been replaced by declarations using Envi's type restriction

```
ENVIRON buffer
 ( data : VAR( FROM[ item$create(...) ] );
   full : VAR( FROM[ boolean$create(...) ] );

   insert : OP( FROM[ item$create(...) ] );
   remove : OP() : FROM[ item$create(...) ];

   PROCESS
     buffer$full := boolean$false;
     DO boolean$true -->
        IN insert(y) WHEN boolean$neg( buffer$full ) -->

                  buffer$full := boolean$true;
                  buffer$data := y

        []  remove() WHEN buffer$full -->

                  buffer$full := boolean$false;
                  RETURN( buffer$data )

        NI
     OD
   END     )
```

Figure 3.13. A Buffer Description.

scheme. The procedures used in the monitor have been replaced by operation declarations and a process that implements the operations. This process repeatedly executes an input statement that handles 'insert' and 'remove' requests. The input guards contain 'WHEN' clauses that ensure that the state of the buffer is appropriate before either an 'insert' or 'remove' request is accepted. These 'WHEN' clauses take the place of the monitor's condition variables. In

addition, the fact that there is only one process receiving 'insert' and 'remove' requests implies that at most one request can be processed at any time. The remainder of the code used in the input commands is similar to the simple statements found in both the cluster and monitor solutions.

While the specification expression in Fig. 3.13 is structurally similar to the monitor shown in Chapter 2, its function is different. The monitor describes a type. Its name can be used to produce elements of this type and representation details of the objects used to represent elements of the type are encapsulated. The specification expression above only describes an object. The expression itself must be evaluated each time such an object is desired, and the representation is not encapsulated. The variable components are as accessible as the operations. The specification expression, however, can be used as part of a buffer type definition.

Like other objects in Envi-0, types are represented using environs. Such an environ is called a type manager. Its components are typically the operations and constants needed to manipulate elements of the type it manages. Often, the most important of these is an operation that acts as a source of all the objects that represent elements of the type. As discussed in the overview, such operations are used in Envi-0 type restrictions to describe the set of elements in the type. The language does not require that any particular name be used for this operation, but following the convention of always using the same name has advantages that will become obvious later. Accordingly, we will always use 'create' as the name of this operation.

In the case of the buffer type, 'create' will be the only operation provided by the type manager. This is because all the other operations needed to manipulate buffers are included as components of the objects themselves. In our definition, this operation will take a type as a parameter and return a buffer that can store elements of that type. The buffer will not do anything to the elements of this type other than store them. Therefore, the only restriction that needs to be placed on create's input parameter is that it be a type manager. This is expressed by the type restriction

```
ENVIRON ( create: OP(...):FROM[ENVIRON()] )
```

A specification expression that describes a type manager for buffers is shown in Fig. 3.14. This description of the type manager includes two descriptions of the buffers themselves. One is the specification expression shown above in Fig. 3.13. This expression describes all of the details of these objects. It is used by the process that implements 'create' to produce instances of the objects it describes.


```
ENVIRON buffer_type
 ( create :
    OP( item: FROM[ ENVIRON
                     ( create: OP(...): FROM[ ENVIRON() ]
                           )  ]
         ): FROM[ENVIRON
                  ( insert:
                      OP( FROM[ item$create(...) ] );
                    remove:
                      OP() : FROM[item$create(...) ]
                     )    ] ;

  PROCESS
    DO boolean$true -->
       IN create( item ) -->  RETURN(

          ENVIRON buffer
            ( data : VAR( FROM[ item$create(...) ] );
              full : VAR( FROM[ boolean$create(...) ] );

              insert : OP( FROM[ item$create(...) ] );
              remove : OP() : FROM[ item$create(...) ];

              PROCESS

                buffer$full := boolean$false;
                DO boolean$true -->
                   IN insert( y) WHEN
                        boolean$neg( buffer$full )

                     --> buffer$full := boolean$true;
                         buffer$data := y

                   []  remove( ) WHEN buffer$full

                     --> buffer$full := boolean$false;
                         RETURN( buffer$data )

                   NI
                OD
              END      )
            )
       NI
    OD
  END      )
```

Figure 3.14. A Single Element Buffer Type.

The other is the specification expression used to describe the return value in the operation declaration for 'create'. It is very different from the first. In particular, it only includes declarations of the operation components. This has the effect of encapsulating the representation of the buffers. In determining the validity of references made to the objects produced by calling 'create', the type system only uses the information provided by the return value type restriction. The fact that the expression used to produce these objects includes more component declarations than are found in this type restriction is ignored. As a result, references to the variable components of objects returned by 'create' are invalid.

The objects described in Figs. 3.14 and 3.14 show how the mechanisms of Envi-0 provide for object creation, dynamic access limitations and static access limitations. The use of a specification expression in the RETURN statement of the 'create' operation illustrates object creation. The input statement that implements 'insert' and 'remove' in Fig 3.14 shows the enforcement of the dynamic access limitations required for correct synchronization. Finally, the type restrictions used in the declarations of the 'create' operations provide the means to enforce the static access limitations needed to encapsulate the definitions.

3.7. Summary

The definition of our basic language is now complete. This language contains our proposals in their simplest form and as such determines the fundamental properties of other languages that can be designed using these proposals. These properties are simple. All data objects belong to one class of objects called environs. An environ is simply a collection of named constant, variable and operational components and unnamed process components. Expressions are used to produce values during execution and to describe sets of values in type restrictions. The specification expression provides the only means to create new objects.

Since all data objects are equivalent in the view of the language, all expressions are valid in any context where any expression is valid, except when limited by component selection. The programmer, on the other hand, typically views the set of data objects as a collection of related subsets, and wishes to require that only expressions producing values in a given subset appear in a given context. The language allows for this by defining an ordering on the set of expressions based on the programmer's declarations. The programmer then specifies the minimal expression that is valid in any context. This system provides for the notions of restriction by source and content used in many other languages. Finally, in our basic language, statements describe the actions performed by the process components of environs. The invocation and input statements are fundamental to our language because they provide for communication between environs.

These features provide all the power needed to solve difficult programming problems. The sparse syntax of the language, however, makes it difficult to use. Accordingly, the next chapter presents a language that is based on the same principles as Envi-0 but provides more syntactic support to the programmer.

Chapter 4
Extending Envi-0

In this chapter a language called Envi is defined. Many of the constructs found in conventional programming languages are provided in Envi. They are defined as abbreviations for certain combinations of facilities available in Envi-0. Thus, while Envi is quite a bit richer than Envi-0 syntactically, the two languages are semantically equivalent.

The presentation of Envi serves two purposes. First, it shows that the concepts underlying our proposals can support mechanisms similar to those found in most current languages. Secondly, it provides a language that can be used to construct examples illustrating the use of the mechanisms we have proposed. As explained in the introduction to Chapter 3, many convenient but nonessential features were omitted from Envi-0 to avoid obscuring the major features of our proposal. Unfortunately, the resulting language is so sparse as to be cumbersome for the construction of interesting sample programs. Envi solves this problem.

The presentation of Envi is divided into two parts. First a series of abbreviations for common uses of Envi-0 constructs is described. These include syntactic extensions providing for infix operator notation, simplified type specifications, common uses of the operation construct, and a special construct for hiding the details of an object's structure. The second part introduces a series of built-in objects provided in Envi that implement common types including the integers, subranges and arrays, and an abbreviation for the definition of enumeration types.

4.1. Abbreviations in Envi

4.1.1. Expression Syntax

The expression syntax used in Envi-0 requires that all component names be fully qualified and that all operation invocations be written in prefix notation. This simple syntax was chosen to avoid adding unnecessary complexity to the language's type checking rules. It makes the language awkward to use, however. For example, the expression

```
4b + a
```

which can be written as

```
4*B + A
```

in most languages becomes


```
integer$add(integer$mult(integer$4,B),A)
```

in Envi-0. To improve this situation, Envi supports a more familiar expression syntax by providing two abbreviations.

The first abbreviation allows qualifying expressions to be omitted if no ambiguity arises. Thus, if 'integer' is the only object with components named 'add', 'mult' or '4', Envi would recognize

```
add(mult(4,B),A)
```

as an abbreviation for the fully qualified expression

```
integer$add(integer$mult(integer$4,B),A)
```

In determining whether an operation name that is not fully qualified is ambiguous, Envi will consider the types of the actual parameters used. On the other hand, Envi will not use information about the context in which an expression appears to determine the correct qualification.

The second form of abbreviation for expressions allows infix operators to be used in place of prefix notation. Envi's scheme for doing this is based on the approach used in Alphard [69]. Envi associates operator symbols with operation names. When a use of the operator symbol is encountered, it is treated as an invocation of the operation whose name is associated with the operator. The Envi operators and their expanded forms are shown in Fig. 4.1. Normal

operator	expanded form
A [B]	A$index(B)$val
A * B	mult(A, B)
A / B	div(A, B)
A + B	add(A, B)
A - B	sub(A, B)
A < B	less(A, B)
A = B	equal(A, B)
A > B	greater(A, B)
~ A	neg(A)
A & B	and(A, B)
A | B	or(A, B)

Figure 4.1. Envi Infix Operator Replacements

precedence rules are used to determine the order in which the operators should be applied. The operators are listed in Fig 4.1 in order of decreasing precedence. Brackets are used for subscripting; the operator '~' is used for negation of both logical and arithmetic values.

Given these abbreviations, expressions like

```
4*B + A
```

can generally be used. Such an expression is transformed into its prefix form,

```
add(mult(4,B),A)
```


by replacing infix operators by the appropriate operation invocation. Then it is recognized as an abbreviation for

```
integer$add(integer$mult(integer$4,B),A)
```

by the provisions for inserting qualifying expressions if no ambiguity arises.

4.1.2. Type Specifications

In the overview of Envi-0 presented in Chapter 3, we observed that the type system was more similar to type systems found in conventional languages than its syntax would suggest. In particular, we identified restriction by source and content as the two basic approaches to type specification used in other languages and showed how they could be used in Envi-0. In Envi, the structure of the type checking system remains the same, but several syntactic extensions are provided to simplify the construction of type restrictions based on restriction by source and restriction by content.

4.1.2.1. Source Restrictions

Restriction by source is the mechanism for type specification used in most conventional languages. That is, when we write

```
VAR x : integer
```

the values that can be stored in 'x' are being described not in terms of their own properties, but in terms of some other object called 'integer'. In most languages, the 'integer' object corresponds roughly to the set of integers. This is the <u>source</u> of all objects that can be stored in 'x'.

In Envi-0, the same restriction can be expressed, but the specification

```
FROM[ integer$create(...) ]
```

must be used. This is because the name 'integer' is not associated with an object corresponding to the set of integers in Envi-0. Instead, it is associated with an object composed of all the basic operations used to process integers. One of these operations will be used to produce the environs that represent members of the type. This operation is the source of integers. The specification

```
FROM[ integer$create(...) ]
```

assumes that 'create' is the name of this operation. The language does not require that the operation name 'create' be used by all type managers, but it proves advantageous to adopt this convention. As a result, most Envi-0 type specifications based on restriction by source are almost identical to this specification for the integers. If 'e' is an expression that identifies the manager for the type, then the restriction used will be


```
FROM[ e$create(...) ]
```

Envi provides an abbreviation for these simple specifications. If Envi encounters an expression, 'e', where a type specification is expected, it uses the restriction

```
FROM[ e$create(...) ]
```

Thus, if 'integer' is placed where a type restriction is expected, the full restriction

```
FROM[ integer$create(...) ]
```

will be assumed.

<u>4.1.2.2. Content Restrictions</u>

The second approach to type specification discussed in Chapter 3 is restriction by content. When this approach is used, an object is described in terms of its own properties, rather than in terms of its producer. Structural equivalence in Pascal[] and the mechanisms used to place restrictions on type parameters to generic definitions in CLU and Alphard are familiar examples of restriction by content.

Restriction by content can be used in Envi by placing a specification expression in a type restriction. Thus, the Envi type restriction shown in Fig 4.2 is essentially equivalent to the CLU WHERE clause

```
FROM[ ENVIRON t
       ( create: OP( ... ): FROM[ ENVIRON() ];
         lt, eq, gt: OP( t, t): boolean
         )  ]
```

Figure 4.2. An Envi-0 Content Restriction.

```
WHERE t HAS lt, eq, gt: PROCTYPE(t,t) RETURNS( bool );
```

Furthermore, this Envi type restriction is not significantly longer or more complicated than the CLU construct, even though it does not use a notation developed specifically for restriction by content. Therefore, it is not necessary to provide an abbreviation for these specifications in Envi. Instead, Envi provides an alternate syntax which adds to the readability of programs by making it more obvious that restriction by content is being used.

The syntax of Envi allows the programmer to replace the header

```
FROM[ ENVIRON ident
```

used in a type restriction by the header

```
OBJECT ident WITH
```


and to omit the closing ']' in such a specification. Thus, the restriction shown above can be rewritten as shown in Fig. 4.3. This makes the programmer's intentions a little more obvious.

4.1.2.3. A Limited Macro Facility

The preceding sections introduced abbreviations for general classes of type specifications. There are also many specific type restrictions that occur frequently enough to warrant abbreviation. In some cases, the specifications involved are unique to a particular program. In others, the specification is used in many programs. For example, the restriction

```
FROM [ ENVIRON () ]
```

plays an important role in encapsulating implementation details. An abbreviation for this type specification would

```
OBJECT t WITH
   ( create: OP( ... ): OBJECT WITH( );
     lt, eq, gt: OP( t, t): boolean
     )
```

Figure 4.3. An Example of Envi's Content Restrictions.

be most useful.

The simplest solution to this need is to provide a mechanism within ENVI to allow the programmer to define names that refer to certain type specifications and to treat occurrences of these names as occurrences of the corresponding type specifications. Names used in this way are called restrictor names.

A programmer introduces a restrictor name through a pseudo-declaration of the form

```
RESTRICTOR ident = type-restriction
```

For example,

```
RESTRICTOR any = FROM[ ENVIRON () ]
```

defines the name 'any' as an abbreviation for the restriction mentioned above.

This construct is described as a pseudo-declaration because, although it appears in the same syntactic context as a component declaration, the name defined does not behave as a component name. A restrictor name is not referenced as a component. A restrictor name can only be used without qualification. Its scope is limited to the body of the environ in which its definition appears.

Restrictor names can be used in contexts where type specifications are allowed. In such contexts, a restrictor name is used by placing the keyword TYPE before it. Thus,

```
TYPE any
```

is a type specification that uses the restrictor name definition shown above.

In the remainder of this thesis, the name 'any' will be used as defined above. In addition, other restrictor names will be defined as examples are considered.

4.1.3. Using Operations

The operation mechanism of Envi-0 is very flexible. It enables the programmer to mimic the behavior of various procedure mechanisms found in other languages including the simple static subroutines of FORTRAN [2], procedures with dynamically allocated activation records as found in Algol [55] and procedures whose calls are subject to mutual exclusion constraints as in a monitor [31]. In addition, it supports more unusual control structures such as coroutines [14,50]. This flexibility is obtained by separating the features usually combined in procedure mechanisms.

There are disadvantages to this approach, however. The code that must be written is often longer than would be required in a language with a specialized construct for the form of procedure being used. In Envi-0, the programmer must combine the use of several language features with the operation to describe even a simple Fortran-like subroutine. Therefore, Envi provides abbreviations for two common uses of the operation construct: the definition of simple mutually exclusive procedures and the definition of procedures with dynamically allocated activation records.

4.1.3.1. Exclusive Operations

The simpler of the two operation abbreviations supports operations whose execution must be mutually exclusive. The solution of the single-element buffer problem shown at the end of Chapter 3 illustrates how such operations are defined in Envi-0. In that example, 'insert' and 'remove' are mutually exclusive. A general way that any set of operations can be made exclusive is by placing their definitions together in one large IN statement enclosed within a nonterminating loop in one process. Fig. 4.4 shows the skeleton of such a definition.

There are two reasonable objections to the code in Fig. 4.4. The first involves the duplicate specification of the operation and parameter names. This information appears once in the component declaration section of the environ and then again in the implementing process. It could be argued that only one specification should be required. On the


```
ENVIRON monitor
  ( a1 : OP( ... type specs ... );
                .
                .
                .
    an : OP( ... type specs ... );
                .
                .   ( other component declarations )
                .
    PROCESS
      DO true -->
          IN a1( ... parm names ... )
               --> ... code of a1
          [] a2( ... parm names ... )
               --> ... code of a2
                        .
                        .
                        .
          [] an( ... parm names ... )
               --> ... code of an
          NI
      OD
    END )
```

Figure 4.4. Mutual Exclusion in Envi-0.

other hand, the component declarations provide a succinct description of the parameterizations of the operations provided by the environ. This improves the readability of programs. Therefore, no change will be made to the language involving these specifications.

The second reasonable objection involves the implementing process. One can argue that the strings of keywords 'PROCESS DO true --> IN' and 'NI OD END' clutter up the program without providing any additional information. So, Envi provides a construct for describing these simple but common processes. In Envi, a component specification of the form

SELECT command_1 [] ... [] command_n END

is equivalent to the process specification

```
PROCESS DO true -->
    IN command_1 [] ... [] command_n NI
OD END
```

Fig. 4.5 shows how the specification of a single element buffer given in Chapter 3 can be rewritten using SELECT and several other abbreviations already introduced in this chapter. SELECT can also be used to abbreviate the description of the implementation of a simple procedure that does not require dynamic allocation of an activation record; in this case it merely contains one input command.

4.1.3.2. Procedures

In languages like Algol, PL/I and Pascal, each procedure call requires the allocation of a new instance of the local variables used by the procedure. This allocation is essential if a procedure is used recursively, if dynamic


```
ENVIRON buffer
 ( data : VAR( item );
   full : VAR( boolean INIT false );

   insert : OP( item );
   remove : OP() : item;

   SELECT
        insert(y) WHEN ~ full

              --> full := true;
                  data := y

   []   remove() WHEN full

              --> full := false;
                  RETURN( data )

   END     )
```

Figure 4.5. An Abbreviated Buffer Description.

array bounds are used, or if multiple calls of the procedure might be made in parallel by independent processes. The invocation of an operation in Envi does not involve such allocation; an invocation is processed in the already existing environment of the process that implements the operation.

While the features for invocation and allocation have been separated in Envi-0, it is possible to use them together to obtain behavior similar to that of a Algol-like procedure. Rather than simply invoking the procedure, the caller must explicitly cause the creation of a new object containing an instance of the procedure's storage and then invoke an operation of the created object. This allocation can be accomplished through the evaluation of an environ specification expression containing a description of the data and code used by the procedure.

Fig. 4.6 contains an Envi description of a recursive factorial procedure. This example will be used to illustrate

```
ENVIRON fact
 ( alloc: OP(): OBJECT WITH
                    ( exec : OP( integer ) : integer;

   SELECT alloc() -->
     RETURN
       ( ENVIRON
          ( exec : OP( integer ) : integer;

            SELECT exec( n ) -->
               IF n = 0   -->
                     RETURN(1)
               [] n > 0 -->
                     RETURN(n*fact$alloc()$exec(n-1))
               FI
            END
            )
          )
   END
  )
```

Figure 4.6. A Recursive Function Definition in Envi-0.

the essentials of procedure definition in Envi-0. The example assumes the existence of the type 'integer'. The details of the integer type will be discussed later in this chapter.

Notice that the recursive call within the factorial definition is not simply

    fact(n-1)

Instead, the expression

    fact$alloc()$exec(n-1)

simulates a standard procedure call. The sub-expression 'fact' accesses the object that implements the factorial function. The use of this object is broken down into two steps. First, the sub-expression

    fact$alloc()

causes the 'alloc' operation of the factorial procedure to allocate a new instance of the local data used to compute factorials. Finally, the invocation of 'exec' in the full expression

    fact$alloc()$exec(n-1)

causes the code of the factorial procedure to be executed in the new data area using the parameter n-1.

Comparing the definition in Fig. 4.6 with the more conventional definition shown in Fig. 4.7 reveals which portions of the Envi factorial specification are unique to the problem and which are part of the standard approach to describing procedures in Envi-0. First, the parameterization of the 'exec' operation depends on the parameterization of the procedure being defined. Also, the body of the input command that implements the 'exec' operation is obtained from the body of the conventional procedure by replacing recursive references to the name 'fact' with the calling sequence 'fact$alloc()$exec'. Everything else in the Envi definition is independent of the procedure being defined. Thus, much can be abbreviated.

In Envi, an environ that implements a procedure can be defined using an expression of the form

```
PROCEDURE fact ( n : integer ) : integer
        BEGIN
           IF n = 0 THEN RETURN( 1)
                    ELSE RETURN( n*fact(n-1) )
        END
```

Figure 4.7. A Conventional Recursive Function Definition.

PROCEDURE id_0(id_1 : $type_1$; ... id_n : $type_n$) : $type_0$
BEGIN statements END

Envi treats such a procedure definition as an abbreviation for the specification expression shown in Fig. 4.8. For example, the factorial would be defined by the component definition shown in Fig. 4.9 using this notation. A further extension to allow a call such as

fact$alloc()$exec(n-1)

```
ENVIRON id0
 ( alloc:
   OP():OBJECT WITH
         ( exec: OP(id1:type1;. . .idn:typen):type0 )
   SELECT
     alloc() -->
       RETURN(
         ENVIRON
          ( exec: OP(id1:type1;. . .idn:typen):type0;
              SELECT
                   exec( id1, ..., idn ) -->
                              statements
              END
             )
                )
   END
 )
```

Figure 4.8. Rule for Expanding Procedure Definitions in Envi.

```
fact : PROCEDURE fact ( n : integer ) : integer
          BEGIN
               IF n = 0  --> RETURN(1)

               [] n > 0 -->
                      RETURN
                         ( n*fact$alloc()$exec(n-1))
               FI
          END
```

Figure 4.9. An Abbreviated Envi Factorial Function.

to be replaced by the simpler call

    fact(n-1)

could be included in the language. It cannot be handled strictly as a local abbreviation, however. It would require that the name 'fact' be recognized as a special name throughout the program.

4.1.4. Encapsulation

In chapter 2, we argued that associating a fixed form of encapsulation with a language's mechanism for object creation limits the flexibility of this mechanism. Accordingly, in Envi-0, no special mechanism for export limitation is included in the specification expression. Instead,

Envi-0's type system provides a means to hide the structure of an object in a way that separates this hiding from the object's creation. The buffer type definition in Chapter 3 shows how this can be done. The examples in the next chapter will illustrate the advantages of this separation. In certain simple examples, however, this separation proves to be a handicap. In this section, we will discuss such an example, and introduce an Envi construct designed to remedy the problem.

Suppose that a programmer wants to use just one single element buffer without taking the time to define a buffer type. In the simplest case, the programmer could simply use the specification expression found within the process that implemented the 'create' operation of the single element buffer type defined in Chapter 3. If the programmer wishes to encapsulate the representation of this single buffer, however, this will not be sufficient. Instead, to hide the components of the buffer, the programmer will have to evaluate the specification expression indirectly by calling an operation that returns the result of evaluating the specification expression. If this is done, representation dependent components of the buffer can be hidden by choosing an appropriate type restriction for the return value of the of the operation used.

Fig. 4.9 shows an expression that uses this technique to hide the variable components of a buffer. While this technique works, it is not a satisfactory solution. The expressions required are too complicated. One would expect the description of a single buffer to be considerably simpler than the description of a buffer type. Instead, the

```
ENVIRON
 ( hide: OP(): OBJECT WITH
                ( insert: OP( item );
                  remove: OP(): item
                  )
   SELECT
     hide() --> RETURN(
                  ENVIRON buffer
                   ( data : VAR( item );
                     full : VAR( boolean INIT false );

                     insert : OP( item );
                     remove : OP() : item;

                     SELECT insert(y) WHEN ~full

                                --> full := true;
                                    data := y

                           []  remove() WHEN full

                                --> full := false;
                                    RETURN( data )

                     END      )
                 )
   END )$hide()
```

Figure 4.9. Encapsulating the Definition of a Environ.

programmer must do most of the work involved in the type definition just to get a single instance.

Fortunately, most of the details of these expressions are the same in all such definitions. The only parts that depend on the particular object being described are the specification expression used to produce the object and the type restrictions given for the return value of the 'hide' operation. This makes it easy to provide an abbreviation for such expressions in Envi.

The construct provided in Envi for encapsulation takes the form

MODULE identifier IS export-list FROM expression

The 'expression' in this construct corresponds to the expression evaluated when the 'hide' operation of the expression in Fig. 4.9 is called. The 'export-list' is an abbreviation for the type restriction used to describe the return value of 'hide' in the figure. In particular, if the type restriction

OBJECT identifier WITH (declaration-list)

would have been used to describe the value produced by 'hide', then 'declaration list' can be used as the 'export-list'. In addition, any component declaration in the export list can be replaced by just the component's name if the declaration is identical to that used in the expression up to substitution of local names. If the identifier could be omitted in the type specification for 'hide' then it can be omitted from the header of the module construct.

Fig. 4.10 shows how the module construct can be used to produce an expression equivalent to that shown in Fig. 9. Other examples of its use will be seen in the description of

```
MODULE IS
   insert; remove
 FROM
   ENVIRON buffer
    ( data : VAR( item );
      full : VAR( boolean INIT false );

      insert : OP( item );
      remove : OP() : item;

      SELECT insert(y) WHEN ~full

                 --> full := true;
                     data := y

           []  remove() WHEN full

                 --> full := false;
                     RETURN( data )

      END     )
```

Figure 4.10. An Example of the Module Construct.

enumeration types later in this chapter.

4.2. Built-in Types

Most programming languages provide a group of built-in data types including at least Boolean, integer and character. In Envi-0, however, only the type Boolean is provided. In principle, this does not weaken the language. All the other common types can be defined within the language. In practice, user defined implementations of the common types would be far too inefficient. They could not compete with built-in types because they could not directly use hardware features such as the arithmetic instructions. Therefore, any practical language based on Envi-0 will have to provide a larger set of built-in types. In recognition of this, the simple types integer, character and facilities for arrays, subranges and enumeration types are provided in the definition of Envi.

In the description of the type Boolean, the notation of Envi-0 was used to specify the structure of the object named 'boolean', even though it could not be used to describe the semantics. This provided a simple way to incorporate information about this type in the type checking system of Envi-0. The same technique will be used in describing the types of Envi. Furthermore, we will either present a complete Envi-0 implementation of each Envi type or explain how one could be constructed. This will be done to show that the presence of these types does not make the semantics of Envi more complicated than that of Envi-0. In addition, it makes the intended semantics of the types precise.

4.2.1. Integers

In Envi, the name 'integer' is assumed to be bound to an object produced by a specification expression of the form shown in Fig 4.11. This specification says that the name 'integer' provides access to an object with components including the integer operations and integer constants. The names 'add', 'sub', 'mult' and 'div' provide access to the arithmetic operations of addition, subtraction, multiplication and division. Their names have been chosen to correspond to the names associated with the appropriate infix operators discussed above. Similarly, 'less', 'greater' and 'equal' provide programmers with access to the usual relational operations on the integers. The extra operators 'succ', 'pred' and 'neg' provide the successor, predecessor and negation operations. Along with these operations, 'integer' is assumed to define the integer constants. Thus the digit 9 is treated as an identifier that selects the appropriate component of 'integer'. An implementation would probably prohibit other uses of these identifiers.

The implementation of the integer operations could be described in Envi-0 by mimicking the hardware


```
ENVIRON integer
    ( create : OP( ... ) : TYPE any;

      add, sub, mult, div :
              OP( integer, integer ) : integer;

      less, greater, equal :
              OP( integer, integer ) : boolean;

      neg, pred, succ :
              OP( integer ) : integer;

      0, 1, 2, 3, 4, 5, ....    : integer$create (...);

      maxint, minint : integer$create(...);

           .
           .
           .

        unspecified process components
           .
           .
           .
      )
```

Figure 4.11. Outline of the Environ 'integer'.

implementation. An integer value would be represented by using one boolean value for each bit used by the hardware. Such objects could be produced by an expression such as

```
    ENVIRON ( b0, b1, b2, ..., b15: VAR( boolean ) )
```

Hardware features could not be used directly to add these integers, however. Instead, each of the basic operations would be described by a long sequence of IF statements that examined the arguments bit by bit.

4.2.2. Enumeration Types

The definition of an enumeration type in Envi-0 is simple, but it can be lengthy. In this section, a definition of the type 'character' is presented as an example of such a definition. Then, a general abbreviation for similar enumeration types is introduced.

4.2.2.1. The Type Character

The type character is provided by assuming that the name 'character' is bound to the object produced by the expression shown in Fig. 4.12.

The members of the 'character' enumeration type are represented as integers. Since a very small set of characters has been chosen for the purposes of this example, only the integers from 0 through 36 are used. The object 'character' provides components defining the character constants 'A', 'B', ... in much the same way that the 'integer' module defined the integer constants. Again, it is reasonable to assume that an implementation would prohibit other uses of these identifiers.

The choice of the integers as an underlying type makes implementation of operations on character values simple. The relationals 'less', 'greater' and 'equal' and the prede-


```
MODULE character IS
  create : OP( ... ) : TYPE any;
  ' '; 'A'; ...'9';
  ord; chr; less; greater; equal; succ; pred
FROM
  ENVIRON character
      ( create : OP( integer ) : integer;

        SELECT
            create(x) WHEN x ≥ 0 & x ≤ 36 --> RETURN(x)
        END;

        ' ' : character$create( 0 );
        'A' : character$create( 1 );
          .
          .
          .
        '9' : character$create( 36);

        less,greater,equal: OP( x,y: character ): boolean;

        succ, pred : OP( character ) : character;

        ord : OP( character ) : integer;
        chr : OP(  integer  ) : character;

        SELECT
               ord(x) --> RETURN( x )

         []    chr(x) WHEN x ≥ 0 & x ≤36 --> RETURN( x )

         []    less(x, y)
                      --> RETURN( integer$less(x, y) )

         []    greater(x, y)
                      --> RETURN( integer$greater(x, y) )

         []    equal(x, y)
                      --> RETURN( integer$equal(x, y) )

         []    succ(x) WHEN integer$less( x,36)
                      --> RETURN( character$create(x+1) )

         []    pred(x) WHEN integer$greater( x, 0 )
                      --> RETURN( character$create( x-1 ))
        END
      )
```

Figure 4.12. The Environ 'character'.

cessor and successor functions are implemented by simply invoking the corresponding 'integer' operations. The 'ord' and 'chr' functions are merely restricted versions of the identity function. Only the type information associated with an object is changed by application of these two operations.

While the use of the integers as a representation type for characters makes implementation of the primitive operations simple, the users of the language should be unaware of or at least unable to make use of this representation. If the specification expression named 'character' used as a subpart of the construct shown in Fig. 4.12 were used alone to describe the type, this would not be the case. The declaration of 'character$create' in the specification expression indicates that

```
character$create(...) ≥ integer$create(...)
```

As a result, every character valued function could be used as an integer valued function. That is,

```
'A' * 'C'/3
```

would be considered type correct.

In Fig. 4.12, the module construct introduced in Section 4.1.4 is used to correctly conceal the use of the integers as a representation type. The export list used in

this module indicates that all components declared in the specification expression should be visible, but it conceals certain details of one component, the 'create' operation. In particular, it changes the type restriction associated with the return value of create in such a way that

```
character$create(...) > ENVIRON()
```

is the strongest statement that can be made about 'character$create' in the type ordering. As a result, only the operations defined in 'character' can be applied to character valued expressions.

4.2.2.2. Other Enumeration Types

The structure of the definition of the character type is not unique. The definition of any enumeration type will involve many of the constructs and techniques used in the definition of 'character'. Accordingly, Envi provides an abbreviation for such definitions.

The syntax for an enumeration type definition will call for the keyword 'ENUM' followed by a list of the identifiers that are to name the elements of the type. Thus, 'char' would be defined by

```
ENUM( ' ', 'A', 'B', ... ,'9' )
```

The meaning of this new construct is defined by a rule to transform such expressions into module expressions. An expression of the form

ENUM(id_1, ..., id_n)

is equivalent to the expression obtained from the expression shown in Fig. 4.13 by replacing the symbols id_1 through id_n with the identifiers used in the enumeration definition.

4.2.3. Derived Types

In addition to the scalar data types, most language provide structured types whose elements consist of collections of objects of more primitive types. Such types include arrays, records and sequences. Many languages also provide subrange types whose elements consist of a subset of the elements of some primitive type.

Structured types and subranges share an important property. They can all be viewed as the result of applying some type producing function to a more primitive type. The production of a subrange type can be seen as the application of some 'subrange' function to an ordered type and two elements of the ordered type. The desired subrange type is the result of this function application. Similarly, an array is produced by applying an 'array' function to an element type and an index type. The result is the desired array type. We will call such types *derived types* because they are derived from simpler types through function application.


```
MODULE enum IS
       create : OP( ... ) : TYPE any
       equal, less, greater,
       succ, pred
       id1, ... , idn,
   FROM
       ENVIRON enum
           ( create : OP( integer ) : integer;

             SELECT
                 create( x ) WHEN x ≥ 0 & x ≤ n-1
                                 --> RETURN( x )
             END

             id1 : create( 0 );
                 .
                 .
                 .
             idn : create( n-1 );

             less, greater, equal :
                     OP( enum, enum ) : boolean;

             succ, pred :
                     OP( enum ) : enum;

             SELECT

                 less( x, y)
                         --> RETURN( integer$less(x,y))

              [] greater( x, y)
                         --> RETURN( integer$greater(x,y))

              [] equal( x, y)
                         --> RETURN( integer$equal(x,y))

              [] succ( x ) WHEN integer$less( x,36)
                         --> RETURN( enum$create( x+1 ) )

              [] pred( x ) WHEN integer$greater( x, 0 )
                         --> RETURN( enum$create( x-1 ) )
             END
    )
```

Figure 4.13. Expansion of an Enumeration Type Definition.

The notion of derived types is particularly important in Envi. This is because in Envi the modules that describe types are data objects themselves. As a result, functions that map from type-defining objects to other type-defining objects can be defined. In this section, definitions of such functions for the production of subranges and arrays are presented.

4.2.3.1. Subrange Types

In Envi, an object named 'subrange' is provided to support the creation of subrange types. This object possesses an operation named 'create' which, when given a type and upper and lower bounds, produces the type definition for a subrange consisting of the elements of the type that fall between the two bounds. The type passed as a parameter to 'subrange$create' is called the base type.

The 'create' operation of 'subrange' cannot be applied to arbitrary base types. In order for the notions of upper and lower bounds to be meaningful the base type must be an ordered type. That is, it must provide relational operations on its elements. Using restriction by content, it is possible to require that the type manager passed to 'create' possess such operations. The restrictor name 'ordered_type' whose definition appears in Fig. 4.14 will be used to do this. It should be noted that this restriction describes the structure of ordered types, but not the semantics. It


```
RESTRICTOR ordered_type =

   OBJECT ordtype WITH
         ( create : OP( ... ) : TYPE any;

           pred, succ : OP( ordtype ) : ordtype;

           less, equal, greater :
                     OP( ordtype; ordtype ) : boolean
         )
```

Figure 4.14. A Type Restrictor for Ordered Type Managers.

is possible to construct an object that meets the structural restrictions but provides operations that do not behave as expected; unfortuately, there is no way the language can preclude this.

The 'create' operation of 'subrange' also produces a type as its value. Restriction by content will also be used in the description of the function's return value. In deciding on the form of the restriction that should be used for the return value, several factors must be considered.

The most important question is whether the elements of the subrange type should be recognized as elements of the base type. Envi provides two alternatives. If the object returned by a call of 'subrange$create' is describe by the return value restrictor associated with 'create' in

```
create : OP( base type ) : TYPE any;
```

then the only information available about objects of the new type will be that they belong to the new type. The subrange and its base type will be completely distinct. If instead, the output of 'subrange$create' is described by the restrictor in

```
create : OP( base_type ) : base_type;
```

then each expression of the new type will also be recognized as an expression of the base type.

The advantage of the second approach is that it allows operations of the base type to be applied to the elements of a subrange. In particular, if the base type is an arithmetic type such as the integers, then the arithmetic operations could be applied to elements of the subrange producing elements of the base type. Therefore, the version of 'subrange' shown below will use this approach.

A second consideration is the choice of the primitive operations that will be provided by the subrange type's manager. The only significant difference between a subrange type and an arbitrary ordered type is that the subrange has upper and lower bounds. Therefore, a subrange type's manager should provide the predecessor, sucessor and rela-

tional operations, together with constant components that provide access to the bounds of the subrange in subrange type managers.

Based on these decisions, the Envi 'subrange_type' is described by the restrictor name defined in Fig. 4.15. Using the restrictors 'ordered_type' and 'subrange_type', the specification of the object named 'subrange' is simple. A possible implementation is shown in Fig. 4.15. It should be emphasized that it is assumed that any implementation of an Envi-like language would provide a low-level implementation of this object. The definition in Fig. 4.16 is intended to precisely describe the behavior of 'subrange'; it is not intended to be the implementation.

```
RESTRICTOR subrange_type =
  OBJECT subtype WITH
        ( create : OP( base_type ) : base_type;

          succ, pred : OP( subtype ) : subtype;

          less, equal, greater :
                     OP( subtype, subtype ) : boolean;

          min, max : subtype$create( ? )
         )
```

<u>Figure 4.15</u>. <u>Restrictor for Subrange Types</u>.

```
ENVIRON subrange
  ( create : OP( base_type : TYPE ordered_type;
                   lower, upper : base_type )
                            : TYPE subrange_type;
    SELECT
       create() -->
         RETURN(
            ENVIRON subtype
              ( create : OP( base_type ) : base_type;

                SELECT
                  create( val ) WHEN val>lower | val=lower
                                   & val<upper | val=upper
                      --> RETURN( val )
                END;

                succ, pred : OP( subtype ) : subtype;

                less, equal, greater :
                          OP( subtype, subtype ) : boolean;

                min : subtype$create( lower );
                max : subtype$create( upper );

                SELECT
                   pred( val ) -->
                      RETURN( create(base_type$pred(val)))
                END;
                SELECT
                   succ( val ) -->
                      RETURN( create(base_type$succ(val)))
                END;
                SELECT
                   less( x, y) -->
                      RETURN( base_type$less( x, y ) )
                END;
                SELECT
                   equal( x, y) -->
                      RETURN( base_type$equal( x, y ) )
                END;
                SELECT
                   greater( x, y) -->
                      RETURN( base type$greater( x, y ) )
                END  )  )
    END )
```

Figure 4.16. The Environ 'subrange'.

4.2.3.2. Arrays

Like subranges, array types in Envi are viewed as derived types. An object named 'array' is built-in to the language. This object provides a 'create' operation that returns a type manager for a new array type when it is applied to type managers for an index type and an element type. For example, the Envi declaration

```
string_type: array$create(integer, character)
```

uses 'array$create' to produce a type manager for the type of arrays of characters indexed by integers. In this section, the behavior of arrays in Envi is explained and an Envi-0 implementation of arrays is outlined.

Fig. 4.17 contains an environ specification expression that describes the structure of the object named 'array'. This object has only one component, the operation create. In the 'array' specification expression the declaration of this operation is preceded by pseudo-declarations for several restrictor names used in the operation's declaration. The restrictor names 'type_def' and 'ordered_type_def' are used to describe the parameters accepted by 'create'. The names 'array_type_def' and 'array_val' are used to describe the object returned by 'create'.

```
ENVIRON array
 ( RESTRICTOR type_def =
      OBJECT WITH
         ( create : OP( ... ) : TYPE any );

   RESTRICTOR ordered_type_def =
      OBJECT ordtype WITH
         ( create : OP( ... ) : TYPE any;
           pred, succ : OP( ordtype ) : ordtype;
           less, equal, greater :
                      OP( ordtype; ordtype ) : boolean
          );

   RESTRICTOR array_type_def =
      OBJECT WITH
         ( create :
              OP( low, high : index_type ) : array_val );

   RESTRICTOR array_val =
      OBJECT WITH
         ( index : OP( index_type ) :
                       FROM [ ENVIRON
                               ( val: VAR( element_type ))
                              ] );

   create: OP( index_type    : TYPE ordered_type_def ;
               element_type : TYPE type_def
                  ): TYPE array_type_def;

   SELECT create( element_type, index_type ) -->
     RETURN(
       ENVIRON array_type
         ( create : OP( index_type, index_type ) :
                                        TYPE array_val;

             SELECT create( low, high ) WHEN high>low -->
               RETURN(
                 ENVIRON array_rep
                  ( index: OP( index_type ):
                         FROM [ ENVIRON
                                 ( val: VAR(element_type))
                                ] );
                    )
                      )
             END ) )
   END )
```

Figure 4.17. The Structure of 'array'.

The first parameter to 'array$create' is the element type. The code that implements an array never manipulates objects of the element type. This code merely manages variables of the element type. As a result, the only requirements placed on this type is that the object that represents it must follow the convention of providing a 'create' operation. Thus, the simplest restriction that can be placed on this object is the restrictor 'type_def' defined in Fig. 4.17. This restrictor will be used again in the next chapter.

The second parameter to 'array$create' describes the index type for the new array type. This expression must satisfy the restrictor 'ordered_type_def' used in the declaration of 'array$create'. This is because, as in the definition of subranges, it must be possible to specify a subset of the index type by providing upper and lower bounds. This can only be done if the type is ordered. The type 'integer', used in the declaration of 'string_type' above, satisfies this restrictor.

The application of 'array$create' to suitable parameters produces a type manager for arrays with the element and index types specified. This object is produced by evaluating the specification expression named 'array_type' in Fig 4.17. It provides a 'create' operation that can be used to produce elements of the type. For example, the expression

```
string_type$create(1,100)
```

will produce an object that represents an array of characters indexed by the integers between 1 and 100.

Array types in Envi are somewhat unusual. Envi treats the index type used as a characteristic of the type, but allows each array in an array type to accept a distinct subrange of values from the index type as indices. Thus, both arrays of characters indexed by the integers 1 through 100 and arrays of characters indexed by the integers 1 through 50 could belong to the type 'string_type'. In most languages, arrays that use different subranges of a type as subscripts are treated as different types. The requirement that a subscript belong to a certain type is different from the requirement that it fall in a certain subrange of the type, however. The first requirement can be enforced statically by appropriate type checking rules. The second must be enforced dynamically. The approach used in Envi recognizes this difference.

Logically, an array is just a function from members of the index type to variables of the element type. In Envi, neither functions nor variables can be returned by an operation. It is possible, however, to return an environ containing a function or to return an environ containing a variable. Therefore, in Envi, an array will be represented by an environ containing an operation that maps from an

index type to environs containing variables of the element type. These objects are produced by evaluating the specification expression named 'array_rep' in Fig. 4.17. This expression includes a declaration for the operation just discussed, but not a process that implements it.

It is possible to implement the operation 'index' within the Envi language. The simplest approach is to represent an array by a list of nodes each containing one of the valid index values and the corresponding array element's value. Then the 'index' operation can be handled by a search through the list to find the node corresponding to the index value used. The details of this implementation are not presented in Fig. 4.17 because any practical implementation of Envi would use standard low-level techniques to provide arrays. The fact that it is possible to implement arrays in Envi, however, is significant, since it assures us that the addition of the object 'array' does not change the nature of the language.

Arrays in Envi function in the same way as arrays in other languages. Given the array type 'string_type' declared above, an array of 120 characters could be bound to the name 'line' by the declaration

```
line: string_type$create(1, 120)
```

Then, the expression

```
line$index(20)$val
```

could be used to access the twentieth element of the array. The abbreviations introduced in Section 1.1 allow this expression to be replaced by the more compact and familiar form

```
line[20]
```

This kind of expression can be used to produce a value or as the target of an assignment. Thus,

```
line[20] := line[21]
```

is a valid assignment in Envi.

4.3. Summary

The definition of Envi is now complete. The extensions that have been made to Envi-0 in this chapter results in a language whose features are more comparable to those of conventional programming languages. At the same time, this new language possesses all the important properties of Envi-0.

As in Envi-0, all data objects in Envi belong to one class of objects called environs. The most important difference between the data domain of Envi and that of Envi-0 is the presence of several important pre-defined objects. These objects provide for the primitive types integer and character, and for derived types including

arrays and subranges.

The set of expressions allowed in Envi is larger than that in Envi-0. While Envi-0 only allowed specification expressions, invocations and identifications, Envi provides many special purpose forms of expression. Special forms of the specification expression are provided for procedures, encapsulated definitions and enumeration types. Invocation expressions are extended to allow infix operators. Nevertheless, expressions are still viewed as general tools for describing sets of objects. Subject to constraints imposed by the type system, any expression type can be used in any context where an expression is expected. Furthermore, expressions are still used to describe sets of objects in type restrictions.

Finally, several special uses of Envi-0's expression based type restrictions have been recognized and special abbreviations for these uses have been provided. While this simplifies the construction of many Envi programs, the expression-based syntax for type specification remains the primary means for describing type restrictions.

Chapter 5

Programming in Envi

In Chapter 2, the goal of designing independent facilities for object creation, static access constraints and dynamic access constraints that would provide the capabilities needed to solve synchonization and data abstraction problems was discussed. This chapter presents examples that show how this is accomplished in the Envi language.

The examples included in the first two sections of the chapter are intended to show how Envi's features provided capabilities equivalent to those provided by the programming constructs discussed in Chapter 2. Section 1 uses the readers/writers problem to examine the use of Envi's features in the solution of synchronization problems. Section 2 shows how Envi's features can be used to define relational types and polymorphic procedures.

The third section contains examples illustrating some of the additional capabilities supported by Envi's mechanisms. The examples shown include the definition of several synchronized queue types. The first of these is unusual because it provides a binary merge operation in addition to the usual insert and remove operations. The second is a

priority queue type which is then used to solve several familiar operating systems problems including the specification of an alarm clock and a disk scheduler. The final example is a queue type used to implement the queues needed in a spooling system. It is unusual because the type manager uses more than one representation scheme for the elements of this type.

5.1. Synchronization in Envi

The features of Envi used in solving synchronization problems are patterned closely after the constructs used in Andrews' Synchronizing Resources (SR) proposal [4]. Most of the arguments and examples used by Andrews in support of his proposals apply to Envi with slight modification. Accordingly, it is not our purpose to present a wide variety of examples showing the use of these features. Instead, we will present examples that show how to handle some of the differences between Envi's synchronization facilities and those of SR, and illustrate those aspects of Andrews' proposals that made them particularly attractive for Envi.

The most important difference between Envi's and SR's synchronization mechanisms involves storage allocation. All modular synchronization constructs depend on the allocation of local storage for each process accessing a shared resource. In the monitor proposal, this allocation corresponds to the instantiation of new instances of the local data areas of monitor procedures when they are called. In SR and several other message-passing based synchronization mechanisms [32,57,63], arrays of processes and message channels enable the programmer to pre-allocate the needed storage. Since Envi is a language particularly concerned with the separation of facilities for the creation of data objects from those for synchronization, a mechanism such as families of processes is undesirable. Instead, in Envi, the same construct used for storage allocation in all other situations, the specification expression, will be used in synchronization problems.

The examples considered in this section are all variations of the readers/writers problem. The solution to one of these problems in SR is discussed by Andrews. His solution employs arrays of both processes and operations. Accordingly, this example provides the opportunity to show how the role of arrays of processes and operations can be filled by other constructs in Envi. In fact, the ease with which the specification expression can fill this role was a factor in the decision to use message-based synchronization in Envi. These examples also provide a brief but strong argument in favor of operation-based synchronization. Bloom [5] has suggested that a good synchronization construct should enable one to clearly separate synchronization constraints from other aspects of an abstraction's implementation. In the solutions we present to the readers/writers

problem this separation is evident. As suggested by Bloom, we present several solutions in which scheduling constraints differ to emphasize this separation.

5.1.1. The Readers/Writers Problem

The readers/writers problem [13,15] is a well known abstraction of the problem of concurrent database access. A group of processes wishes to share a data base. Each process may either read from or write to locations in the data base. Several reads may be processed in parallel, but whenever one process is writing, no other process may be allowed to write or read. The variations of the problem involve different scheduling schemes that all satisfy this basic exclusion requirement. As Andrews has observed, many solutions to this problem [9,13,15,31] provide the means by which processes *can* follow the exclusion rules but do not *force* the processes to follow them. Our solutions will enforce these rules. This is done by encapsulating the details of the database so that it can only be accessed through 'read' and 'write' operations that include the needed synchronization code.

Encapsulating the database so that all accesses are made through two operations requires the use of storage allocation features in both Envi and SR. The problem is that, in these languages, one process handles all calls made to a given operation. This feature of the operation mechanism makes it difficult to implement an operation like 'read'. It is impossible for several processes to read in parallel if there is only one process handling all 'read' requests. In SR the problem is solved by defining 'read' as a array of operations so that read requests from different processes are handled by different processes. In Envi, there is only one process that receives 'read' requests, but it creates new processes to handle these requests so that such requests can be processed in parallel.

Fig. 5.1 shows how such a 'read' operation is implemented in Envi. This module is not yet a solution to the readers/writers problem, however -- it does not enforce any synchronization constraints. The details required to enforce these constraints will be added later. They have been omitted here to emphasize the way in which the parallelism required by the 'read' operation is supported.

When a user invokes the 'read' operation defined in Fig. 5.1, an environ specification expression is used to create an object containing a process that will perform the requested read. This new object is returned to the user. To obtain the value read from the database, the user invokes the 'get' operation provided by the object that 'read' returns. Thus, the expression

```
read(i)$get()
```



```
MODULE rw IS
   read; write;
FROM
  ENVIRON RW
    ( read: OP( i: integer ):
                                 OBJECT WITH( get: OP(): item);
      write: OP( v: item; i: integer ):
                                 OBJECT WITH( wait: OP());

      database: array$create(item,integer)$create(1,n);

      SELECT
          read(i) -->
             RETURN( ENVIRON
                        ( get: OP(): item;
                          output: VAR( item );

                          PROCESS
                             output := database[i];
                             IN get() --> RETURN(output)
                             NI
                          END )
                         )

      [] write(v, i) -->
             RETURN( ENVIRON
                        ( wait: OP();

                          PROCESS
                             database[i] := v;
                             IN wait() --> SKIP NI
                          END )
                    )
      END )
```

Figure 5.1. Encapsulation of Data Base Accesses.

produces the value of record 'i'. The 'write' operation is implemented and used in a similar way.

The effort involved in creating a process for each read seems ridiculous when all that the created process does is execute a single assignment statement. Similarly, the effort taken to encapsulate the details of the data base may seem useless if the data base is a simple array. The reader should remember that this simple view of the data base is part of the abstraction. In a realistic situation each access of the data base might require several disk accesses. In such situations, both the programming effort required for encapsulation and the overhead involved in process creation would be justifiable.

In the following three sections, this module will be used as a skeleton to construct several solutions to the readers/writers problem that use different scheduling policies while enforcing the essential exclusion constraints. These examples show that the technique used to implement the 'read' and 'write' operations is flexible enough to support the specification of complex synchronization schemes.

5.1.2. A Readers Preference Solution

This section shows how the module presented in the preceding section can be completed by adding code that enforces the exclusion contraints required to solve the readers/writers problem. The solution is shown in Fig. 5.2. While this solution does not explicitly seek to enforce any scheduling policy, the steps required to enforce the

exclusion constraints lead to what is called a weak reader's preference solution. That is, readers who arrive while the data base is being read are allowed to proceed, even if a writer is waiting.

The exclusion constraints are enforced by adding four synchronization operations to the original specification. In addition, the code of the processes that implement the actions of reading and writing have been modified to invoke these new operations. The 'read' processes all invoke 'startread' before executing the statements that access the data base and 'endread' when the access is complete. Similarly, the 'write' processes invoke 'startwrite' before each access and 'endwrite' after each access. The advantage of adding synchronization code in this way is that it obviously has no effect on the correctness of the code that actually accesses the data base.

In this solution, the implementation of the four synchronization operations depends on an integer variable, 'state'. This variable's value is equal to the number of active readers if reads are in progress, zero if the data base is idle, and minus 1 if a write is in progress. The correctness of the implementation of the synchronization operations follows obviously from this interpretation of 'state'. All 'startread' requests are delayed when 'state' is negative; thus, no new reads can start once a write is in

```
MODULE rw IS
   read; write;
FROM
  ENVIRON RW
    ( read: OP( i: integer ):
                          OBJECT WITH( get: OP(): item);
      write: OP( v: item; i: integer ):
                          OBJECT WITH( wait: OP());
      startread, startwrite, endread, endwrite : OP();

      state: VAR( integer INIT 0);
      database: array$create(item,integer)$create(1,n);

      SELECT read( i) -->
             RETURN( ENVIRON
                        ( get: OP(): item;
                          output: VAR( item );
                          PROCESS
                             startread();
                             output := database[i];
                             endread();
                             IN get() -->
                                   RETURN( output )
                             NI
                          END )     )

      [] write ( v, i) -->
             RETURN( ENVIRON
                        ( wait: OP();
                          PROCESS
                             startwrite();
                             database[i] := v;
                             endwrite();
                             IN wait() --> SKIP NI
                          END )      )

      [] startread() WHEN ~(state$val() < 0)    -->
             state := state$val() + 1;

      [] endread() --> state := state$val() - 1

      [] startwrite() WHEN state$val() = 0    -->
             state := -1

      [] endwrite() --> state := 0

      END )
```

Figure 5.2. A Reader's Priority Readers/writers Solution.

progress. Furthermore, a 'startwrite' is only processed when the data base is idle; thus, no other reads or writes can be processed while a write is being processed.

5.1.3. Writer's Preference

A simple alternative to the reader's preference solution is to force readers to wait if a writer is waiting. This can be done by keeping a count of the number of writers who are waiting and refusing to accept 'startread' requests whenever this count is positive. This approach is called a writer's preference solution. Such a solution is shown in Fig. 5.3.

An important feature of this solution is that the code that accesses the data base and those parts of the input guards that ensure exclusion (i.e. uses of 'state') have not been modified. The changes in scheduling may affect the solution's performance, but not its correctness. A second observation is that the technique of creating servant processes to handle user requests in parallel makes it easy to count the number of requests being processed. Adding the statement

```
swrite := swrite + 1
```

before the creation of writer processes and the statement

```
swrite := swrite - 1
```

```
MODULE rw IS
   read; write;
FROM
  ENVIRON RW
    ( read: OP( i: integer ):
                         OBJECT WITH( get: OP(): item);
      write: OP( v: item; i: integer ) :
                         OBJECT WITH( wait: OP());
      startread, startwrite, endread, endwrite : OP();

      state, swrite: VAR( integer INIT 0);
      database: array$create(item,integer)$create(1,n);

      SELECT read( i) -->
             RETURN( ENVIRON
                        ( get: OP(): item;
                         output: VAR( item );
                         PROCESS
                            startread();
                            output := database[i];
                            endread();
                            IN get() -->
                                  RETURN( output )
                            NI
                         END )       )
      [] write ( v, i) -->
             swrite := swrite+1;
             RETURN( ENVIRON
                        ( wait: OP();
                         PROCESS
                            startwrite();
                            database[i] := v;
                            endwrite();
                            IN wait() --> SKIP NI
                         END )        )
      [] startread() WHEN ~(state < 0)  &
                         swrite = 0 -->
             state := state + 1

      [] endread() --> state := state - 1;

      [] startwrite() WHEN state = 0    -->
             state := -1

      [] endwrite() --> state := 0;
                        swrite := swrite - 1
      END )
```

Figure 5.3. A Writer's Priority Readers/writers Solution.

to 'endwrite' accomplishes this. In the solution, this variable is used to delay readers whenever one or more writers is waiting.

5.1.4. Alternating Readers and Writers

The final solution considered here is based on Andrews' [4] SR solution of the problem. This solution causes readers and writers to alternate if both are waiting. The Envi version is shown in Fig. 5.4. It uses three variables to schedule requests. 'Swrite' and 'sread' are counters that indicate how many writers and readers are currently waiting for access to the database. 'Writerlast' is a Boolean variable that indicates whether a reader or a writer was last allowed to use the database.

This solution shows precisely how the features of SR that have not been included in Envi can be replaced by Envi features. It also shows again how the implementation of scheduling policies and synchronization requirements can be kept independent from one another and from the specification of the basic algorithms used to manipulate the object being described. Here, the variables 'swrite', 'sread' and 'writerlast' are used for scheduling. Their addition does not affect the used of the variable 'state', which is used to enforce the exclusion constraints, or the use of the array 'database'.

```
MODULE rw IS
   read; write;
FROM
  ENVIRON RW
   ( read : OP( i : integer ):
                      OBJECT WITH( get : OP() : item);
     write: OP( v : item; i : integer ) :
                      OBJECT WITH( wait: OP());
     startread, startwrite, endread, endwrite  : OP();
     state, sread, swrite : VAR( integer INIT 0);
     writerlast : VAR( boolean INIT false);
     database : array$create(item,integer)$create(1,n);

     SELECT read( i) -->  sread := sread + 1;
            RETURN( ENVIRON
                     ( get : OP() : item;
                       output : VAR( item );
                       PROCESS
                          startread();
                          output := database[ i];
                          endread();
                          IN get() -->
                                RETURN( output )
                          NI
                       END )      )
     [] write ( v, i) -->  swrite := swrite + 1;
            RETURN( ENVIRON
                     ( wait : OP();
                       PROCESS startwrite();
                               database[ i] := v;
                               endwrite();
                               IN wait() --> SKIP NI
                       END )      )

     [] startread() WHEN ~ (state < 0) &
            ( writerlast | swrite = 0 ) -->
          writerlast := false; sread := sread - 1
          state := state + 1;

     [] startwrite() WHEN state = 0 &
            ( ~ writerlast | sread = 0 ) -->
          state := -1;   writerlast := true;
          swrite := swrite - 1

     [] endread() --> state := state - 1;
     [] endwrite() --> state := 0
     END )
```

Figure 5.4. Alternating Readers and Writers.

5.2. Simple Types and Polymorphism

In Chapter 2, we discussed the differences between the data object definition facilities provided by most synchronization languages and those found in data abstraction languages. Synchronization proposals, like the monitor, tend to support the definition of objects that are accessed through their own components. On the other hand, data abstraction languages generally provide for the definition of classes of objects that are accessed through the components of a separate object that we have called the type manager. The examples of the preceding section showed how the facilities of Envi can be used to define objects similar to those defined using the facilities of synchronization proposals. In this section, we show how Envi can be used to support the approach of the data abstraction languages.

One strength of the approach used in the data abstraction languages is that it supports the definition of types with binary operations. Accordingly, the first example in this section is the definition of a type that involves several binary operations, the rational numbers.

Another feature found in data abstraction languages is the ability to define polymorphic procedures. Envi also supports the definition of such procedures. To illustrate this, a polynomial evaluation procedure is presented as the second example.

5.2.1. The Rational Numbers

The rational numbers provide an excellent example of a type definition in Envi. The details of the definition are simple, but the problem is not trivial. As a result, it illustrates the features of the language without requiring an inordinately large program. The definition includes specifications for the binary arithmetic operations of addition, subtractions, multiplication and division, as well as the relational operations.

One interesting aspect of the problem involves the choice of a representation for the rational numbers. It is obvious that a rational can be represented as a pair of integers corresponding to the numerator and denominator of a fraction. However, each rational can be represented by an infinite number of such pairs.

Generally, storage limitations require the elimination of all representations of a rational that correspond to a fraction that is not in lowest terms. If this is not done, operations on the rationals tend to produce fractions with extremely large numerators and denominators. Accordingly our definition will always divide the numerator and denominator of a fraction by their greatest common divisor before using them to represent a rational.

Even when all fractions not in lowest terms are eliminated, there are still two representations for each rational number since

$$\frac{a}{b} = \frac{-a}{-b}$$

An implementation has four choices. It can allow either representation, require a positive denominator, require a positive numerator, or require that both numerator and denominator be positive and keep the sign of the rational as a separate boolean value. The first two possibilities are the best. The first allows new rationals to be created quickly; the second allows for more efficient implementation of the relational operations. Our implementation uses the first.

A definition of the type 'rational' is shown in Fig. 5.5. Our decisions about representation are completely expressed by the definition of the 'create' operation. 'Create' takes any pair of integers as input and treats them as the numerator and denominator of a fraction. Using Euclid's algorithm, it computes their greatest common divisor and returns an environ with two components 'num' and 'den'. 'Num' and 'den' correspond to the components of a fraction that is in lowest terms and equal in value to the fraction corresponding to create's inputs.

```
MODULE IS create: OP( x, y: integer ): TYPE any;
        add; sub; mult; div; equal; less; greater;
FROM
 ENVIRON rational
  ( create: OP( x, y: integer ):
                OBJECT WITH( num,den:integer$create(...));
    add , sub, mult, div:
                OP( x, y: rational ): rational;
    equal, less, greater, comp:
                OP( x, y: rational ): boolean;
    a, b : VAR( integer );

    SELECT create( x, y) WHEN ~ (y = 0) -->
            a := abs(x); b := abs(y);
            IF a > 0 -->
                  DO a < b  --> b := b - a

                  [] a > b  --> a := a - b

                  OD
            [] ~(a > 0) -->  SKIP
            FI;
            RETURN( ENVIRON ( num: x/b;
                              den: y/b  ) )
    END;

    SELECT add ( x, y) -->
        RETURN( rational$create( x$num*y$den+y$num*x$den,
                                 x$den*y$den ) )
    END;
    SELECT sub ( x, y) -->
        RETURN( rational$create( x$num*y$den-y$num*x$den,
                                 x$den*y$den ) )
    END;
    SELECT mult ( x, y) -->
        RETURN( rational$create( x$num*y$num, x$den*y$den ) )
    END;
    SELECT div ( x, y) WHEN ~ (y$num = 0) -->
        RETURN( rational$create( x$num*y$den, x$den*y$num ) )
    END;
    SELECT comp( x, y) -->
        RETURN( (x$num*y$den - y$num*x$den)*x$den*y$den)
    END;
    SELECT equal  (x, y) --> RETURN( comp(x,y) = 0 ) END;

    SELECT less   (x, y) --> RETURN( comp(x,y) < 0 ) END;

    SELECT greater(x, y) --> RETURN( comp(x,y) > 0 ) END
    )
```

Figure 5.5. The Type 'rational'.

The other operations provided by 'rational' access the 'num' and 'den' components of the environs used to represent rationals. The arithmetic operations all compute the numerator and denominator of a fraction whose value equals the result desired and then call 'create' to produce a representation in lowest terms. The relationals are implemented by subtracting one value from another and then examining the sign of the result. In order to determine the sign of the result, its numerator and denominator are multiplied together since the result of this multiplication will have the same sign as the original fraction. The operation 'comp' is used to perform this computation. Note that the operations are described by separate SELECT constructs; this allows them to execute in parallel.

The definition is encapsulated using the module construct. Here, the module hides two features of the underlying definition. First, the omission of the name 'comp' from the export list prevents access to this function from outside the module. More importantly, the export list entry for 'create' modifies the description associated with this operation by replacing the return value type specification with the specification

```
TYPE any
```

This prevents access to the 'num' and 'den' components of rational values from outside the module.

5.2.2. A Polymorphic Procedure

The existence of type managers makes it possible to write definitions that are applicable to elements of different types, as long as the set of primitive operations used by the operations implementation are shared by all the types to which the operation will be applied. This is done by making the operation access the implementations of the primitive operations that it uses through a type manager that is passed to it as a parameter. In Envi, this is particularly easy since type managers are simply environs.

The operation of evaluating a polynomial at a point can be defined in this way. To define such an operation, one must choose a representation for polynomials. We assume that any programmer who wishes to use the polynomial evaluator will place the coefficients of the polynomial in an array and pass the array and its length as parameters. The type associated with the polynomial's coefficients must provide the operations of addition and multiplication, but no other detail of the type's definition is used by the operation. A definition of a polynomial evaluation procedure based on these assumptions is shown in Fig 5.6.

As an example, suppose that a programmer has defined an array

```
coefs: array$create(rational,integer)$create(1,3)
```



```
polyval:
 ENVIRON
  ( RESTRICTOR polyrep =
           FROM[ array$create(T,integer)$create(...) ];

    RESTRICTOR elmnt_type =
           OBJECT type WITH
              ( create: OP( ... ):  TYPE any;
                add, mult: OP( x,y: type ): type );

    create:
      OP( T: TYPE elmnt_type ):
         OBJECT WITH
            ( eval: OP( poly: TYPE polyrep;
                        n: integer, x: T ): T  ) ;

    go: OP( T: TYPE elmnt_type; poly: TYPE polyrep;
                             n: integer; x: T ): T;

    SELECT
      create( T ) -->
        RETURN(
          ENVIRON
           ( eval: OP( poly: TYPE polyrep;
                       n: integer, x: T ): T;

             ptr: VAR( integer);
             temp: VAR( T);

             SELECT eval( poly; n; x ) -->

               ptr := n-1;
               temp := poly[ n ];

               DO ptr > 0 -->
                  temp := temp * x ;
                  temp := temp+poly[ ptr ];
                  ptr  := ptr - 1
               OD ;
               RETURN( temp )
             END )
             )
    END;

    SELECT
      go( T, poly, n, x) -->
       RETURN( polyval$create(t)$eval(poly,n,x) )
    END )
```

Figure 5.6. A Polymorphic Polynomial Evaluation Procedure.

and has initialized it by executing the statements

```
coefs[ 1 ] := rational$create(1,2)
coefs[ 2 ] := rational$create(3,7)
coefs[ 3 ] := rational$create(4,3)
```

Then, the expression

```
polyval$go(rational, coefs, 3, rational$create(6,1) )
```

could be used to evaluate the polynomial

$$\frac{4}{3} x^2 + \frac{3}{7} x + \frac{1}{2}$$

at the value 6.

Examination of the operation 'go' used in the preceding example reveals that it does not perform the evaluation itself. Instead, 'go' is used to hide the actual calling sequence for evaluating polynomials from the user. When 'go' is called, it first calls the 'create' operation of 'polyval'. This operation uses a specification expression to produce a new object that provides an operation called 'eval'. This operation is then called to actually evaluate the polynomial.

The two levels of calling used in this expression are needed because Envi separates allocation from invocation. Evaluation of a polynomial requires the use of a temporary

variable of the same type as the coefficients of the polynomial. When a call to 'go' is made it is processed in the already existing environment of the process that receives it. This process cannot contain variables for every possible type of polynomial coefficient. Therefore, it must create a new environment that contains a temporary variable of the appropriate type and then pass the task of evaluating the polynomial to this new environment. 'Go' creates the new environment by passing the type of the coefficients to the 'create' operation. It then asks the new environment to evaluate the polynomial by invoking its 'eval' operation.

5.3. Queues

In the preceding sections, we have shown how Envi's features can be used to mimic the features of other data abstraction and synchronization proposals. The strength of the language, however, springs from the fact that Envi's independent mechanisms can be combined in ways not possible with the special purpose constructs found in these other proposals. In this section, we concentrate on such combinations. This is done by considering the definition and use of several kinds of queues.

5.3.1. A Mergeable FIFO Queue

We first describe a FIFO queue designed for a scheduling problem suggested by Brinch Hansen's Conditional Critical Region proposal [7]. A conditional critical region is a statement of the form

```
REGION v DO BEGIN ... AWAIT b ... END
```

associated with a shared variable 'v'. The letter 'b' represents a boolean expression involving only the components of 'v'. The implementation of the statement must ensure that at most one process is allowed to execute in any of the regions associated with 'v' at any time. If a process encounters the statement 'AWAIT b' when the value of 'b' is false, it leaves the region, permitting other processes to enter it. When the region is available, a process suspended because of an AWAIT may reenter the region and attempt to continue by re-executing the AWAIT statement.

Brinch Hansen suggests implementing this construct by associating two queues with each shared variable. When a process attempts to enter a region it is placed on the main queue associated with v, Q_v. When the region is available and Q_v is non-empty, a process is removed from Q_v and allowed to proceed. If the process executing in the region is suspended because of an AWAIT statement, it is added to an event queue, Q_e. When a process completes the execution of the region, all processes in Q_e are moved to Q_v to be given another chance to execute the AWAIT statement.

If a queue type providing only the usual 'insert' and 'remove' operations is used to solve this problem, moving the elements on Q_e to Q_v will require as many 'insert' and 'remove' requests as there are elements on Q_e. This is unnecessarily costly. The merge operation can be performed in constant time if each queue is represented as a linked list and a special 'merge' operation is provided. The merge operation will simply copy the head pointer of Q_e to the last element of Q_v and then set the tail pointer of Q_v equal to the tail pointer of Q_e.

While the linked list implementation can easily solve the efficiency problem, another difficulty remains. 'Merge' is a binary operation. Its implementation requires access to the underlying details of the representations of two data objects. This suggests that the approach of the data abstraction languages is most appropriate to the definition of the type to which 'merge' belongs. On the other hand, 'insert', 'remove' and 'empty' are all operations that act on a single object and all these operations require synchronization. This suggests that the approach of the synchronization languages is most appropriate to the definition of this queue type. Thus, the definition of this type does not fit neatly into either of the two major schemes for type definition.

This conflict is resolved by Envi's ability to combine the two approaches to data type definition. With respect to the 'insert', 'remove' and 'empty' operations, the types definition will follow the style of the monitor. These operations will be components of the data objects themselves rather than components of the type managers. The 'merge' operation will follow the other approach. It will be specified as a component of the type manager.

The definition of this queue depends on a record type whose elements will be used as the nodes of the linked list that represents the queue. To avoid adding details to the definition of the queue itself, this types definition is presented separately in Fig. 5.7. Like the definition of other derived types that we have considered, the node type definition has two levels. The outer level provides a 'create' operation that returns instances of types. The inner level describes the types produced by the outer 'create'. Accordingly, it defines another 'create' operation that produces the nodes used to represent queues.

Using this definition of the node type, a complete definition of a mergeable queue type is shown in Fig. 5.8. Like the node type, the mergeable queue is a derived type. As a result, the 'create' operation provided by the object defined returns a queue type rather than a queue. Examination of this operation's declaration makes the distinction


```
ENVIRON qnodes
 ( RESTRICTOR node =
      OBJECT WITH
        ( data: element type$create(...);
          is_last: VAR( boolean INIT true);
          next: VAR( node_type ) );

   create: OP( element-type: TYPE type_def ):
                OBJECT node_type WITH
                    ( create: OP( element_type ): TYPE node );

   SELECT create( element_type ) -->
     RETURN(
       ENVIRON node_type
        ( create: OP( element_type ): TYPE node;

          SELECT make_node (val) -->
             RETURN( ENVIRON
                       ( data : val;
                         is_last : VAR( boolean INIT true);
                         next : VAR( make_node)
                         )
                     )
          END  )
          )
  END )
```

Figure 5.7. A Node Type for Queues.

```
ENVIRON q_type
 ( create: OP( t: TYPE type_def ):
             OBJECT queue WITH
              ( create: OP( ): OBJECT WITH
                               ( insert: OP( t );
                                 remove: OP(): t;
                                 empty: OP(): boolean
                                 );
               merge: OP( main, add: queue ): queue );

  SELECT create( t ) -->
    RETURN(
      ENVIRON queue
       ( nodes: node_type$create( t );

         create: OP(): OBJECT WITH
                      ( insert: OP( t );
                        remove: OP(): t;
                        empty:  OP(): boolean;
                        lock, unlock: OP();
                        head, tail: VAR( nodes );
                        is_empty: VAR(boolean INIT true));

         merge: OP( main, add: queue ): queue;

         SELECT merge( qv, qe ) -->
            qe$lock();
            IF ~qe$is_empty -->

                 qv$lock();
                 IF qv$is_empty -->
                       qv$head := qe$head;

                 [] ~qv$is_empty -->
                       qv$tail$next := qe$head;
                       qv$tail$is_last := false;
                       qv$tail := qe$tail

                 FI;
                 qe$is_empty := true;
                 qv$is_empty := false;
                 qv$unlock()

            [] qe$is_empty -->
                 SKIP

            FI;
            qe$unlock();
            RETURN( qv )
         END
```

Figure 5.8. A Mergeable Queue Type (cont. on next page).


```
    SELECT create( ) -->

      RETURN(
        ENVIRON
         ( insert: OP( t );
           remove: OP(): t;
           empty:  OP(): boolean;
           lock, unlock: OP();

           output: VAR( t );
           head, tail: VAR( nodes );
           is_empty, free: VAR( boolean INIT true );

           SELECT
              insert ( input ) WHEN free -->

              IF is_empty -->
                 head := nodes$create( input );
                 tail := head
              [] ~is_empty  -->
                 tail $ next :=
                        nodes$create( input);
                 tail $ is_last := false;
                 tail := tail $ next
              FI

           [] remove ( ) WHEN free & ~is_empty -->

              output := head $ data;

              IF head $ is_last -->
                 is_empty := true
              [] ~head $ is_last -->
                 head := head $ next
              FI;
              RETURN( output )

           [] empty ( ) WHEN free -->

              RETURN( is_empty )

           [] lock ( ) WHEN free --> free := false

           [] unlock ( ) --> free := true
           END )
         )
    END  ) )
END )
```

Figure 5.8 (cont.). A Mergeable Queue Type.

between 'merge' and the other operations provided with queues obvious. It indicates that 'create' produces an object with two operation components named 'create' and 'merge'. This object is the type manager for a mergeable queue type. The 'insert', 'remove', and 'empty' operations are not components of this object. Instead they are components of the objects produced by the type manager's 'create' operation. Thus, if 'a' and 'b' are queues, an insert is performed by selecting a component of 'a' or 'b', as in

```
a$insert( ... )
```

A merge is performed by selecting a component of the type manager and then providing 'a' and 'b' as parameters as in

```
queue$merge( a, b)
```

Just as 'merge' is referenced differently than the other operations, its implementation references the details of each queue's representation in a way that is different from that used by operations like 'insert'. Within the body of the input command that implements 'merge', the parameter names 'main' and 'add' are known to be bound to objects that behave as if produced by the expression

```
queue$create(...)
```

Because of the description of the operation 'create' in

'queue', such objects are known to have 8 components, including the variables 'head' and 'tail'. Accordingly, within the input guard, the linked lists representing the queue parameters are accessed by expressions such as

```
main$head
```

and

```
add$tail
```

The operation 'insert' on the other hand takes no queues as parameters. Instead, because the input command that implements 'insert' occurs within the specification expression used to create queues, it can access the components of the queue whose 'insert' component was selected when the operation was invoked. These references depend only on the specification expression, not on the return value restriction associated with 'create'.

Once the details of the type system that allow the two different kinds of operations to access the representations of queues are understood, the details of the implementation are quite simple. The variables 'head' and 'tail' point to the first and last elements of the linked list representing the queue. 'Is_empty' is true when all nodes have been removed from the queue. The operations 'insert', 'remove' and 'empty' use these variables in the obvious way. 'Lock' and 'unlock' are operations used by 'merge' to prevent access to a queue involved in a merge operation. Like the variables used, these locking operations are hidden from the users of the queue type by the description of the operation 'q_type$create'. A merge is accomplished by locking both queues, moving the nodes of 'add' to the front of the queue 'main', and then unlocking the queues. Special attention must be given to the case of the empty queue by all these operations.

5.3.2. Priority Queues

In this section, an Envi definition for a priority queue type is given and used to implement two familiar operating system components, a timer interface and a disk head scheduler. These example illustrate several advantages of Envi's mechanisms. The definition of the priority queue type itself shows that the two level definition scheme used in all the derived type definitions shown thus far is not the only way to define such types in Envi. The two examples show both the usefulness of the technique of creating servant processes to handle parallel requests and the flexibility of Envi's encapsulation mechanisms.

5.3.2.1. The Queue Definition

A definition of priority queue type is shown in Fig. 5.9. The 'create' operation returns an instance of a prior-


```
ENVIRON prio_q
 ( RESTRICTOR prio_elmnt_type =
       OBJECT WITH
            ( create: OP( ... ):
                          OBJECT WITH
                            ( key: integer$create( ... ))  );

   RESTRICTOR p_queue = OBJECT WITH
                           ( insert: OP( T );
                             remove: OP(): T;
                             empty : OP(): boolean;
                             minkey: OP(): integer );

   create : OP( T: prio_elmnt_type ) :
                  OBJECT WITH ( create: OP(): TYPE p_queue );

   SELECT create( T ) -->
     RETURN(
       ENVIRON
        ( create: OP(): TYPE p_queue;

           SELECT create() -->
             RETURN(
               ENVIRON
                ( insert: OP( T );
                  remove: OP(): T;
                  empty : OP(): boolean;
                  minkey: OP(): integer;

                  minpair: OP(integer): integer;

                  size: VAR( integer INIT 0);
                  q: array$create(T,integer)$(1,n);

                  ptr: VAR( integer);
                  output: VAR( T );

                  SELECT minpair( x ) -->
                       IF q[x]$key > q[x+1]$key
                          & x+1 < size
                                          --> RETURN(x+1)
                       [] ~ (q[x]$key > q[x+1]$key
                              & x+1 < size )
                                          --> RETURN(x)
                       FI
                  END;
```

Figure 5.9. A Type of Priority Queues (cont. on next page).

```
          SELECT
             insert( x: T) WHEN size < n -->

             ptr := size + 1;

             DO ptr  > 1 & q[ptr/2]$key > x -->
                   q[ ptr ] := q[ ptr/2 ];
                   ptr := ptr/2
             OD;

             q[ ptr ] := x;
             size := size + 1

          [] remove( ) WHEN size > 0 -->

             output := q[ 1 ];
             ptr := minpair( 2 );

             DO ptr < size & q[ptr]$key < q[size]$key -->
                   q[ ptr/2 ] := q[ ptr ];
                   ptr := minpair( 2*ptr )
             OD;

             q[ ptr/2 ] := q[size];
             size := size - 1;
             RETURN( output )

          [] empty() --> RETURN( size = 0 )

          [] minkey() --> RETURN( q[1]$key )
          END
         ) )
      END ) )
  END )
```

__Figure 5.9__ (__cont.__). __A Type of Priority Queues__.

---

ity queue. The definition requires that each element of the type whose members are to be stored in the queue have a integer component named 'key'. This component's value is used as the priority of the element.

The objects returned by 'create' provide four operations. 'Insert' and 'empty' perform the obvious functions. 'Remove' returns the element with the smallest priority value in the queue. 'Minkey' returns the minimum priority value associated with any element in the queue. The objects produced also possess other components, but these are hidden from the user by the restrictor 'p_queue'.

Like any complex data structure, a priority queue can be implemented in several ways. Our definition uses a heap. The heap that implements the queue is stored in the array 'q'. The root is kept at 'q[1]'. The sons of node 'q[n]' are stored in nodes 'q[2*n]' and 'q[2*n+1]'. The implementation ensures that the sons of each node have priority values greater than or equal to their father's. 'Size' is a variable containing the number of elements currently in the queue.

'Insert' requests are processed by incrementing 'size' and then shifting down one position those elements of the heap on the path from 'q[size]' to the root whose priority values are greater than that of the inserted element. The new element is placed in the opening left by this shift.

'Remove' saves the element at the root and then shifts elements up toward the root until an element whose priority is greater than that of the element at 'q[size]' is found. The element at 'q[size]' is moved into the hole left by this shift and then size is decremented. Finally, the value that had been at the root is returned. 'Minpair' is an auxiliary operation used by 'remove' to determine which of the sons of a node should be shifted up. It selects the son with the smaller priority value.

One interesting feature of this definition of priority queues is that it does not use the two level structure that has been used in the definition of other types in this chapter. The 'create' operation of the object 'prio_q' actually returns a queue, while the 'create' operations associated with all the other derived types considered have produced type managers that in turn produced the objects desired.

This difference is significant because is shows that Envi does not support one fixed notion of type definition. It merely provides tools that enable programmers to construct objects that behave as they believe types should behave. The two level structure used in most of our examples produces objects that best reflect our notion of 'type'. This does not imply that others might not prefer alternate approaches or that we might not prefer alternate approaches under special circumstances.

The advantage of the technique used in the specification of 'prio_q' is that it produces shorter definitions and is easier to use. Thus, 'prio_q' is shorter than it would

be if the usual approach had been used, and a particular priority queue can be produced directly by evaluating

```
prio_q$create(task)
```

without taking the intermediate step of producing a type manager for the subtype of 'task' priority queues.

There are, however, disadvantages associated with this approach. To illustrate this, consider the problem of creating an array of priority queues of tasks. To do this, the type manager for priority queues of tasks must be passed to the 'array$create' operation. Unfortunately, such a type manager does not exist here. All that exists is the type manager for all priority queues. Therefore, such an array cannot be created unless an appropriate type manager is programmed. Here, there is no need to define an array of priority queues. Envi allows us to take advantage of this fact and use a simpler method of type definition.

5.3.2.2. An Alarm Clock

In many applications, processes need to delay their execution until some particular time in the future. For this reason, the hardware of most machines provides a clock that can be set to interrupt the system. Unfortunately, there is normally only one such hardware clock and yet there may be many processes who wish to use it. The problem is to simulate many clocks in software using the single hardware clock.

The first step in specifying a software clock is the selection of a software model of the hardware clock. The environ 'clock' defined in Fig. 5.10 will be used for this purpose. It is based on the clock comparator feature of the IBM 370 architecture [34]. One memory location, 'time', holds the current time, while another, 'setting', holds the time at which an interrupt should be generated. A third variable, 'enabled', has a boolean value. An interrupt can only be generated if its value is 'true'. The two processes in the environ mimic the actions that the hardware performs on these variables. The first process increments the

```
clock: ENVIRON
    ( time, setting: VAR( integer INIT 0);
      enabled: VAR( boolean INIT false );

      PROCESS
        DO true --> time := time$val() + 1 OD
      END;
      PROCESS
        DO true -->
          IF enabled & setting > time --> ring() FI
        OD
      END )
```

Figure 5.10. A Model for the Hardware Clock.

variable 'time' at regular intervals. The second repeatedly compares the current time with 'setting' and generates an interrupt when appropriate by calling 'ring'.

using this environ, a software clock is defined in Fig. 5.11. It provides an operation 'delay' that a process can invoke with an integer value 't' in order to be delayed for 't' time units. Like the readers/writers solutions, this object must be able to process several requests in parallel. This is again done by creating servant processes to handle requests. Here, a type of timer request servers called 'tserv' is defined. Its members are objects containing the servant processes. Like the processes used in the readers/writers problems, the members of the type 'tserv' provide an operation 'wait' whose invocation will not be completed until the request has been completely processed. For 'delay', this means that the 'wait' operation will not complete until the interval has passed. Thus, the statement

```
delay(100)$wait()
```

would be used to delay execution for 100 time units.

The alarm clock is implemented using a priority queue of tserv's. Each of the tserv's contains a component named 'key' whose value is equal to the time at which an invocation of its 'wait' operation should complete. This 'key' is used as its priority value. Therefore, the servant process

```
MODULE IS
  delay;
FROM
  ENVIRON timer
   ( delay: OP( integer ): OBJECT WITH
                              ( wait: OP() );
     clock: ENVIRON
       ( time, setting: VAR( integer INIT 0);
         enabled: VAR( boolean INIT false );
         PROCESS
           DO true --> time := time + 1   OD
         END;
         PROCESS
           DO true -->
             IF enabled & setting > time --> ring() FI
           OD
         END );

     tserv: ENVIRON
       ( create: OP( integer ):
                      OBJECT WITH
                        ( key: integer$create(...);
                          go, wait: OP() );
         SELECT create( t ) -->
            RETURN( ENVIRON
                      ( key: t;
                        go, wait: OP();
                        PROCESS IN go() --> SKIP NI;
                                IN wait() --> SKIP NI
                        END )  )
         END );

     queue: prio_q$create(tserv)$create();
     output: VAR( tserv );

     SELECT delay( t ) -->
         output := tserv$create(t+time);
         queue$insert( output );
         setting := queue$minkey();
         enabled := true;
         RETURN( output )

     [] ring() -->
         queue$remove()$go();
         IF queue$empty() --> enabled := false
         [] ~queue$empty() --> setting := queue$minkey()
         FI
     END )
```

Figure 5.11. An Alarm Clock.

whose wait operation should be allowed to complete first will be at the head of the priority queue.

Whenever the queue of tserv's is non-empty, the clock's setting is made equal to the value of the minimal key in the priority queue. As a result, when the clock interrupts, it must be time to let the 'wait' operation of the first object on the queue complete. The implementation of the 'ring' operation does this by removing this object from the queue and invoking its 'go' operation. The clock is then disabled or reset depending on the state of the priority queue.

5.3.2.3. A Disk Scheduler

In this section, the priority queue type defined in Section 3.2.1 is used to solve the problem of scheduling disk head motion. This problem has been used as an example in the presentation of both the Monitor and the SR language (as well as other proposals). Our solution differs from these in that it distinguishes between 'read' and 'write' requests rather than assuming a single 'perform disk i/o' request. It is somewhat more difficult to coordinate the scheduling of two operations with different implementations, than to schedule requests for a single operation. In particular, this added detail shows how Envi's type system allows differences between the server processes handling 'read' and 'write' requests to be hidden from parts of the scheduler. Unfortunately, this makes our solution somewhat longer than the simpler disk schedulers presented in other language proposals.

Several disk scheduling policies have been proposed and analyzed [62]. Our solution uses the algorithm called SCAN, in which the disk head makes passes back and forth across the disk surface, changing direction only when there are no more pending request ahead.

The disk scheduler presented below assumes that the physical disk can be accessed as if it were an Envi environ with the structure described in Fig. 5.12. The function 'dread' accepts a disk address in the form of a cylinder number and a record number and returns the value found at that disk address. The operation 'dwrite' performs a disk write when invoked with a disk address and a value of type 'data' to be written.

```
disk: ENVIRON
  ( dread: OP( cyl, recd: integer ): data;
    dwrite: OP( cyl, recd: integer; data );
              .
              .
              .   )
```

Figure 5.12. _A Model for a Hardware Disk Unit_.

The specification of a disk scheduler in Envi is shown in Fig. 5.13. It describes an object that provides two operations, 'read' and 'write'. As in the previous example, parallel processing of requests is provided by creating objects containing servant processes. Therefore, the 'read' request does not return a value of type 'data'. Instead, it returns an object containing the servant process for the request. This object provides a 'wait' operation that returns the desired value when it becomes available. Thus,

```
read(cyl,recd)$wait()
```

is an expressions that might be used to perform a read. Write requests are processed in a similar manner.

In the definition of 'diskio', the 'read' and 'write' operations are declared together with the operations 'rserv' and 'wserv'. These latter operations are important parts of the implementation of 'read' and 'write'. They are used within 'diskio' to create the objects that are eventually returned to the user by the 'read' and 'write' operations. Each of these objects contains a servant process that handles a single request. It first waits for the scheduler to invoke its 'go' operation, indicating that it is now its turn to use the disk. The process then invokes the appropriate operation of the 'disk' environ. Finally, it informs the scheduler that it is done by calling 'iodone' and informs the user that it is done by accepting an

```
MODULE diskio IS read : OP( cyl,recd: integer ):
                            OBJECT WITH
                                ( wait: OP(): data );
                 write: OP( cyl,recd: integer; val: data ):
                            OBJECT WITH
                                ( wait: OP() );
FROM
  ENVIRON diskio
    (  read, rserv: OP( cyl,recd: integer ):
                          OBJECT WITH
                            ( key: integer$create(...);
                              wait: OP(): data;
                              go: OP() );
     write, wserv: OP( cyl, recd: integer; val: data ):
                          OBJECT WITH
                            ( key: integer$create(...);
                              wait, go: OP() );
     schedule, iodone: OP();

     disk: ENVIRON
       ( dread: OP( cyl, recd: integer ): data;
         dwrite: OP( cyl, recd: integer; data );
                         . . .       );
     RESTRICTOR server =
            OBJECT WITH( key: integer$create(...);
                         go: OP() );
     dserv: ENVIRON
         ( create: OP( TYPE server ): TYPE server;
           SELECT create( b ) --> RETURN( b ) END );
     up, down: prio_q$create( dserv );
     headpos: VAR( integer INIT 0);
     rtemp: VAR( FROM[ rserv(...) ] );
     wtemp: VAR( FROM[ wserv(...) ] );
     temp: VAR( TYPE server );
     isup: VAR( boolean );

     SELECT rserv( cyl, recd ) -->
         RETURN(
           ENVIRON
             ( key: cyl;
               wait: OP(): data;
               go: OP();
               output: VAR( data );
               PROCESS
                 IN go() --> SKIP NI;
                 output := disk$dread( abs(cyl), recd);
                 iodone();
                 IN wait() --> RETURN( output ) NI
               END )      )
```

Figure 5.13. A Disk Request Scheduler (cont. on next page).


```
[] wserv( cyl, recd, val ) -->
    RETURN( ENVIRON
             ( key: cyl;
               wait, go: OP();
               PROCESS
                 IN go() --> SKIP NI;
                 disk$dwrite( abs(cyl), recd, val);
                 iodone();
                 IN wait() --> SKIP NI
               END )     )
END;

SELECT read( cyl, recd) -->
         IF cyl < headpos -->
              rtemp := rserv(~cyl,recd);
              down$insert( dserv$create(rtemp) )
         [] ~ cyl < headpos -->
              rtemp := rserv(cyl,recd);
              up$insert( dserv$create(rtemp) )
         FI;
         schedule(); RETURN( rtemp )
[] write( cyl, recd, val ) -->
         IF cyl < headpos -->
              wtemp := wserv(~cyl,recd,val);
              down$insert( dserv$create(wtemp) )
         [] ~ cyl < headpos -->
              wtemp := wserv(cyl,recd,val);
              up$insert( dserv$create(wtemp) )
         FI;
         schedule(); RETURN( wtemp )
[] iodone() --> busy := false; schedule()
END;
SELECT schedule() -->
   IF ~ busy -->
     IF ~ ( down$empty() & up$empty() ) -->
        IF (isup & ~up$empty()) | down$empty() -->
               isup := true;
               temp := up$remove
        [] (~isup & ~down$empty()) | up$empty() -->
               isup := false;
               temp := down$remove
        FI;
        headpos := abs(temp$key);  temp$go()

     [] down$empty() & up$empty() --> busy := false
     FI
  [] busy --> SKIP
  FI
END )
```

Figure 5.13 (cont.). A Disk Request Scheduler.

invocation of the 'wait' operation.

The behavior of these objects makes it easy to schedule disk requests. The scheduler maintains one priority queue, 'up', of requests to be processed as the head travels up to higher cylinder numbers and another, 'down', of requests to be processed in the other direction. The 'read' and 'write' operations simply add the objects produced by 'rserv' and 'wserv' to the appropriate queue. Then, they notify the scheduler that a new request has been added by invoking 'schedule'.

Whenever 'schedule' is invoked while the disk is inactive and the queues are non-empty, it removes the next object from the queue corresponding to the head's current direction and invokes its 'go' operation. It then updates the 'headpos' variable which is kept equal to the number of the cylinder at which the head is positioned. If the queue corresponding to the current direction is empty, the direction of motion is reversed, and the other queue is examined.

As the previous discussion indicates, the objects produced by 'rserv' and 'wserv' have many similarities. They are, however, far from identical. The wait operation in objects produced by 'rserv' returns a value while the 'wait' operation in those produced by 'wserv' does not. The objects produced by 'rserv' have an extra component named 'output'. The code within the 'rserv' objects calls 'dread'

while the code within 'wserv' objects calls 'dwrite'. Finally, they are produced by different operations. These differences are important, because the scheduling of disk requests is accomplished by placing all these objects on two priority queues of the same type. Our priority queue manager requires a type as input when a queue is created and only allows objects of that type to be stored on the queue. Somehow, the objects produced by 'rserv' and 'wserv' must be made to belong to the same type despite their differences.

In Envi, it is simple to join these two groups of objects to form a single type. An object, named 'dserv', is defined that provides the create operation that will function as the generator of elements of the type. This 'create' operation does not produce new objects. It is merely a restricted identity function. It will take objects produced by 'rserv' or 'wserv' and return them unchanged, except in the eyes of the type system. By doing this, it encapsulates information about the important common features of the objects produced by 'wserv' and 'rserv'. The restrictor 'server' used in the definition of 'dserv$create' enables us to inform the type system that all these objects share the components 'go' and 'key'. These are the only components needed by the priority queue type or the scheduler, so this is very useful information.

This use of 'dserv' is just one of the ways in which objects produced by 'rserv' and 'wserv' are manipulated through the type system. Consider the use of an object produced by 'rserv' in this program. As originally produced and stored in the variable 'rtemp', this object has four components. By passing the object through the 'dserv$create' operation two components are hidden. This ensures that the code of the queue manager and the scheduler which removes these objects from the queue can only access the components 'go' and 'key'. Furthermore, the definition of the queue manager, shown earlier in this section, hides the 'go' component so that its code can only see 'key'. Finally, when the object is returned to the user, the type restriction associated with 'read' in the export list hides all its components but 'wait'. Thus, in this single example, these objects are viewed in four different ways in different contexts.

5.3.3. A Spooling System

A spooling system normally uses several queues. There is a queue of jobs that have been input and need to be processed. There is a queue of jobs that need to be output. In addition, the jobs on these first two queues are themselves queues of input lines and output lines. In this section, we define a FIFO queue type and then use it to provide

all of the queues needed in the description of a spooling system.

Even a simple FIFO queue can be implemented in several ways. If a bound can be placed on its length, an array can be used to hold the elements with pointers indicating the head and tail positions. If a storage allocator is available and no bound can be placed on the queue's length, a linked list of dynamically allocated nodes can be used to implement the queue. Alternatively, some of the advantages of each of these simple methods can be combined by forming a linked list of blocks that each contains a fixed number of queue entries. This last "blocked" implementation is the most likely to be used for the queues in a large spooling system.

The queue type presented in this section is unusual because it supports all three of the implementations just described. Normally, the implementor of a queue type would select one implementation to be used in the representation of all elements of the type. If different implementations were deemed necessary in different parts of a program, distinct types would be defined. Instead, the queue type shown below allows the user to select the appropriate implementation when a queue is created. All the queue instances produced are members of the same type even though they may use different implementations.

The definition for this queue type is shown in Fig. 5.14. This definition supports FIFO queues as elements of derived types. Thus, the object described provides a 'create' operation, which returns a queue type when applied to the type manager for the element type. Each of the objects produced in this way provides its own 'create' operation, which returns a queue when invoked with parameters that describe the implementation to be used. The three kinds of implementations provided are a bounded queue that uses an array as a circular list, a simple linked list with one queue element per node, and a linked list in which each node stores many queue elements.

Most of the text of the definition is concerned with the implementation of the 'create' operation that produces queues. This operation takes two parameters. The first is an element of the enumeration type 'queue$kind' that indicates which of the three implementations to use. The second is a size parameter that determines the size of the array used in the bounded implementation and the size of the blocks used in the blocked implementation. Each call to the 'create' operation is processed by executing an IF statement with one branch for each of the three implementations supported. In each branch, a specification expression is evaluated to produce an object that implements a queue in the appropriate way and then this object is returned to the caller.


```
ENVIRON queue
 ( kind: ENUM( bounded, long, blocked );

   create:
       OP( element_type: TYPE type_def ): TYPE queue_type;

   SELECT create( element_type ) --> RETURN(

     ENVIRON
      ( create:
           OP( queue$kind; integer ): TYPE typed_queue;

        SELECT create ( option, size ) -->

          IF option = bounded -->
            RETURN(
              ENVIRON
               ( insert: OP( element_type );
                 remove: OP( ): element_type;
                 full  : OP( ): boolean;
                 empty : OP( ): boolean;

                 queue :
                    array$
                      create(integer,element_type)$
                                      create( 0, size-1);

                 head, tail, full: VAR( integer  INIT 0 );

                 SELECT
                    insert(input) WHEN full<size -->
                      full := full + 1;
                      head := mod((head + 1), size);
                      queue [ head ] := input

                 [] remove () WHEN full > 0 -->
                      full := full - 1;
                      tail :=
                          mod((tail + 1), size);
                      RETURN( queue [ tail] )

                 [] full () -->
                     RETURN( full = size )

                 [] empty () --> RETURN ( full=0 )

                 END  )       )
```

Figure 5.14. A Multi-implementation Queue Type (continued on next page).

```
[] option = blocked -->
  RETURN(
    ENVIRON seg_queue
      ( insert: OP( element_type );
        remove: OP(): element_type;
        full  : OP(): boolean;
        empty : OP(): boolean;
        seg_gen: queue$create( element_type);
        seg_queue : queue$
                       create(seg_gen)$
                              create( long, 0);
        head, tail: VAR( seg_gen);
        is_empty: VAR( boolean  INIT true);

        PROCESS
          tail := seg_gen$create(bounded, size);
          head := tail;
          DO true -->
            IN insert( input) -->

               IF tail $ full() -->
                   tail :=
                      seg_gen$create( bounded,size);
                   seg_queue$insert( tail )
               [] ~ tail $ full -->
                   SKIP
               FI;
               tail $ insert( input) ;
               is_empty := false

            [] remove () WHEN ~ is_empty -->

               output := head $ remove ();
               IF head$empty() -->
                   IF seg_queue$empty() -->
                       is_empty := true
                   [] ~ seg_queue$empty -->
                       head := seg_queue$remove()
                   FI
               [] ~ head$empty() -->  SKIP
               FI;
               RETURN( output )

            [] empty() --> RETURN( is_empty )

            [] full() --> RETURN ( false )
            NI
          OD
      END  )          )
```

Figure 5.14 (cont.). A Multi-implementation Queue Type
(continued on next page).


```
[] option = long -->

  RETURN(
    ENVIRON queue
      ( insert: OP( element_type);
        remove: OP( ): element_type;
        full  : OP( ): boolean;
        empty : OP( ): boolean;

        nodes: node_type$create( element_type );
        output: VAR( element_type);
        head, tail: VAR( nodes );
        is_empty: VAR( boolean INIT true);

        SELECT insert ( input ) -->

          IF is_empty -->
              head := nodes$create( input );
              tail := head

          [] ~ is_empty -->
              tail$next := nodes$create(input);
              tail$is_last := false;
              tail := tail$next
          FI

        [] remove ( ) WHEN ~ is_empty -->

            output := head $ data;

            IF head $ is_last -->
                    is_empty := true
            [] ~ head $ is_last -->
                    head := head $ next
            FI;
            RETURN ( output )

        [] empty ( ) --> RETURN( is_empty )

        [] full ( ) --> RETURN ( false )

        END )
        )

      FI
    END create )
          )
ND create )
```

Figure 5.14 (cont.). A Multi-implementation Queue Type.

The first branch of the IF statement is taken when the user calls 'create' with 'bounded' as its first parameter. The specification expression used here implements a queue by treating the elements of an array as a ring. The variables 'head' and 'tail' point to the last elements inserted and removed respectively. The variable 'full' counts the number of items in the queue. To simplify the expression, the modulus operator 'mod' is assumed to be an operation of type 'integer'.

At the other extreme, the third branch of the IF statement returns an object using a purely linked implementation of the queue. The details of this implementation will not be discussed here because they are identical to that used for mergeable queues in Section 5.3.1.

Finally, if the second branch of the IF statement is executed, an object using a blocked implementation of the queue is returned. A blocked queue is represented by a linked queue of bounded queues. The name 'seg-queue' is declared to be a queue of queues implemented as a linked list. The elements placed in 'seg_queue' are queues of elements implemented using arrays. Thus, this third implementation uses the other two.

The variable 'tail' refers to the last bounded queue of elements. When an insertion is made into a blocked queue, it is processed by inserting the element into the bounded

queue 'tail', unless this queue is full. If 'tail' is full, it is inserted into 'seg_queue' and a new, empty bounded queue is assigned to 'tail'.

The variable 'head' is used similarly. When a remove is made from the blocked queue, it is processed by removing an element from the bounded queue 'head'. Then, if 'head' is empty, a new bounded queue is removed from 'seg_queue' and assigned to 'head'.

Now, as an example of the use of this queue type, we present an implementation of a simple spooling system. Because our language does not attempt to address the issue of machine level I/O, we will assume that someone has defined environs readerl, ... , readern corresponding to the hardware readers. This could either be part of the language's implementation, much like the definition of 'integer', or the result of a 'device environ' feature, similar to the concept of a device module in MODULA [67]. However they are produced, each of these reader objects will satisfy the restrictor

```
RESTRICTOR reader = OBJECT WITH ( read: OP(): cards )
```

where 'cards' is a type defined by

```
cards : array$create(character,integer)
```

Similarly, we assume predefined names printer1, ..., printerm corresponding to the hardware printers. Each of these objects satisfies the restrictor

```
RESTRICTOR printer = OBJECT WITH ( write: OP( lines );
                                   eject: OP() )
```

where 'lines' is declared by

```
lines : array$create(character,integer)
```

The queues used by the system will all be blocked queues. The input queue will be defined by

```
in_queue: queue$create(in_jobs)$create(blocked, size)
```

In this definition, 'in_jobs' is used as the name of a type defining the actual jobs to be queued. Each job will just be a file of cards. The card reader drivers will insert cards into such files and the job processors will remove cards. Therefore, these job files just behave as queues of cards. The blocked queue already provides an appropriate implementation for such files. So, 'in_jobs' can be defined as

```
in_jobs: queue$create(cards)
```

Note that while 'in_queue' is bound to a single queue, 'in_jobs' is a type of queues.

Similarly, we can define an output queue that allows the job processors to send objects to printer drivers. For this we must define a type 'out_jobs' whose elements will be queues of 'lines'. The definitions are:

```
out_jobs: queue$create(lines);
out_queue: queue$create(out_jobs)$create(blocked,size)
```

Next, a type, 'in_driver', that allows us to create a reader driver for each physical reader is defined in Fig 5.15. The device drivers defined by this type declaration read cards from the reader until they find an end of file character. When this occurs, they put the completed job on the input queue and begin building a new input job. With this definition, the readers can be activated by executing:

```
in_driver $ create ( reader1 ) ;
        .
        .
        .
in_driver $ create ( readern ) ;
```

The output drivers are very similar to the input drivers. Their type definition is shown in Fig. 5.16. Given this definition, output drivers are activated by statements of the form:

```
in_driver :
    ENVIRON
     ( create: OP( reader ): TYPE any;

       SELECT create( device ) -->
         RETURN(
           ENVIRON
            ( cur_card: VAR( cards );
              cur_job : VAR( in_jobs );

              PROCESS
                cur_job :=
                     in_jobs$create(blocked,sizel);

                DO true -->
                   cur_card := device $ read;
                   cur_job $ insert ( cur_card );

                   IF cur_card [1] = eofchar -->

                      in_queue $ insert ( cur_job);
                      cur_job := in_jobs $
                                create( blocked,sizel)

                   [] else -->
                      SKIP
                   FI

                OD
             END )
              )

       END  )
```

Figure 5.15. A Reader Process Type.


```
out_driver :
    ENVIRON
     ( create: OP( printer ): TYPE any;

       SELECT create( device ) -->
         RETURN(
           ENVIRON
            ( cur_job: VAR( out_jobs);

              PROCESS
                DO true -->
                   device $ eject();
                   cur_job := out_queue$remove();

                   DO ~ cur_job$empty () -->
                       device$print(cur_job$remove())
                   OD
                OD
              END )
               )
       END  )
```

Figure 5.16. A Printer Driver Process Type.

```
out_driver $ create( printeri );
```

Finally, we can give a skeletal definition of an executor type. This definition is shown in Fig. 5.17. Now, the entire system can be made active by one or more activations of the statement:

```
job_processor $ create ();
```

```
job_processor :
  ENVIRON
   ( create: OP(): TYPE any;

     SELECT create() -->
       RETURN(
         ENVIRON
          ( cur_in:  VAR( in_jobs);
            cur_out: VAR( out_jobs);

            PROCESS
              DO true -->

                 cur_in := in_queue$remove;
                 cur_out :=
                      out_jobs$create(blocked,size2);

                  {  process the job - using
                        cur_in $ remove() - to read cards
                        cur_out$insert(x) - to print  }

                 out_queue $ insert ( cur_out );
              OD
            END )
             )
     END  )
```

Figure 5.17. An Executor Process Type.

As specified, this system only uses the 'blocked' implementation of queues. In our opinion, this would probably be the best approach. If one wished to use a different implementation for some of the queues, however, the multi-implementation queue type makes it easy. For example, on might argue that since the number of jobs in the system is

likely to be small compared to the size of the jobs, the input and output queues should be linked. This can be accomplished by replacing the declarations for 'in_queue' and 'out_queue' by

```
in_queue: queue$create(in_jobs)$create(linked,size)
out_queue: queue$create(out_jobs)$create(linked,size)
```

No other changes would be required.

Another variation that is made possible by the multi-implementation queue type is a system in which different queue implementations are used for different jobs. For example, if on some system, a particular reader was used only for a class of jobs known to contain less than 200 cards, these jobs might best be stored in a 'bounded' queue. This could be done by creating job files for this reader with the statement

```
cur_job := in_jobs$create(bounded,200);
```

while continuing to create job files for other readers with the statement

```
cur_job := in_jobs$create(blocked,size1);
```

In fact, the queue implementation to be used might be included as a parameter to the operation that creates reader drivers. Doing this would require no other changes. Because both the bounded and the blocked job files belong to the same type, they can all be inserted into the 'in_jobs' queue.

Chapter 6

An Examination of the Features of Envi

The identification of the innovations made by a new language proposal is often difficult. Strange syntax can make common features unrecognizable, while familiar syntax can hide radically new mechanisms. The goal of this chapter is to aid the reader in identifying and evaluating the innovative features of Envi.

As in most languages, much of the Envi language is not new. The statement structure of the language follows a proposal of Dijkstra [20]; the features for procedural abstraction are based on Andrews' operations [4]; and most of the expression syntax is standard. Two features of Envi are unusual, however. The first is the data space of the language. The second involves its use of expressions.

The data space of Envi is unusual because all objects in it are similar: they are all environs. As the examples have shown, this permits a few language mechanisms to provide a great deal of programming power. Section 6.1 examines how this approach differs from those used in other languages and discusses its advantages.

Two aspects of the use of expressions in Envi are also unusual. The first is use of the specification expressions as a means to create new data objects. The second is the use of expressions as the basis of type specifications. These innovations are examined in Section 6.2.

6.1. The Environ

The data space of Envi is based on a single type of data object, the Environ. Each environ is merely a collection of components. The concept of such collections was introduced to programming as early as the definition of COBOL [1]. They have been provided as record types by almost every language since Pascal [66]. Other proposals have allowed the definition of aggregate objects more sophisticated than the record. The Simula 67 class [16], the monitor [31], the CLU cluster [46] and the Ada package [57] are just a few examples. There are, however, two facts about the environ that make it unusual. The first is that all data values in an Envi program are environs. The second is that any object can be used as a component of an environ.

By data values, we mean those objects that can be produced by expressions, passed as parameters and assigned to variables. In Envi, only environs are used as data values. This does not mean that simple arithmetic values, arrays, procedures and other objects cannot be used in Envi. Instead, environs are used to represent elements of types

that are normally considered simple, while objects such as processes and operations are components of environs.

The second unusual feature of the environ is the range of component types allowed. Many languages limit the types of components that can be included in aggregate objects or distinguish between classes of aggregates based on their components types. For example, CLU distinguishes between simple records and clusters, which are aggregates whose components include procedures. The environ is an attempt to provide a single notion of aggregation that applies whether the components are bits, functions or concurrent processes.

In the following sections we explore the differences between the Envi notion of aggregate objects and those in other languages.

6.1.1. Classes

Almost all synchronization and data abstraction proposals, including Envi, trace their heritage to the Simula 67 class [16]. The environ and the class are quite similar. Both allow objects to be constructed with components that include variables and procedures. Both allow any of an aggregate's components to be accessed by selection. Both allow aggregate values to be passed as parameters and assigned to variables, although Simula uses explicit pointer values to reference classes while Envi's pointers are implicit. In fact, the main differences between environs and classes are not differences between the object themselves, but differences in the ways that they interact with other language features.

To understand these interactions, consider the sample Simula code shown in Fig 6.1. This code defines a class of objects that represent binary search trees. There are several distinct objects involved in this definition. First, there is the syntactic construct itself. This is called a <u>class declaration</u>. This construct leads to the production of two distinct types of objects. The first is called a <u>class</u>. In the example, this object is bound to the name 'tree'. It serves a manager for the type being defined. In particular, it can be used as a template to produce elements of the type, which are called <u>objects of the class</u>. This is accomplished by evaluating an expression of the form

```
NEW tree(x)
```

The name of a class can also be used to declare variables that can hold pointers to objects of the class and to specify that a parameter to a procedure will be an object of the class.

ıhe objects of the class 'tree' are aggregates with five components: the parameter 'val', the local variables


```
CLASS tree(val); INTEGER val;
  BEGIN
     REF(tree) left, right;

     PROCEDURE insert(x); integer x;
       BEGIN
          IF x < val THEN
                IF left == NONE
                      THEN left :- NEW tree(x)
                      ELSE left.insert(x)
          ELSE
                IF right == NONE
                      THEN right :- NEW tree(x)
                      ELSE right.insert(x)
       END insert;

     REF(tree) PROCEDURE find(x); INTEGER x;
       find :-
          IF x = val
               THEN THIS tree
               ELSE
                 IF x < val
                     THEN
                       ( IF left == NONE
                              THEN NONE
                              ELSE left.find(x)  )
                     ELSE
                       ( IF right == NONE
                              THEN NONE
                              ELSE right.find(x) ) ;

  END tree
```

<u>Figure 6.1. A Simula Implementation of Trees</u>.

'left' and 'right', and the procedures 'insert' and 'find'. Each of the objects of the class represents a node of a tree. The parameter 'val' holds the value stored at the node represented by a particular object. The variables 'left' and 'right' store pointers to the objects representing the left and right sons of the node. The procedure 'insert' is used to add values to the subtree of which the node is the root. The procedure 'find' searches the subtree, returning the object representing the node at which the value sought is found.

Whereas the environ is the only kind of data value in Envi, Simula distinguishes between objects of classes and objects belonging to the built-in types of the language. Elements of the two groups are manipulated in different ways. Objects of built-in types are manipulated through operations that are applied to them. Objects of classes are manipulated through components that belong to them. In some cases, operations performed with identical notations have different semantics. For example, assignment of simple values involves the copying of values while assignment of objects of classes involves the copying of pointers.

The mechanisms for using classes are also different from those for using objects of classes. This distinction is not very significant in Simula, because few mechanisms are provided for manipulating classes. However, when this approach to type managers is extended to languages where they can be passed as parameters to polymorphic procedures and generic definitions, the distinction becomes very significant. The complexity of the resulting languages is multiplied by the addition of these new mechanisms. The results of such extensions are considered in the following sections.

6.1.2. Generalized Records

By the term generalized record, we refer to aggregate data objects that resemble records in that they consist of components, but differ from records in that they allow procedures and types, as well as variables, as components. This category includes the Mesa definition module [22], the Euclid module [41] and the Ada package [57]. Here, we are interested in the differences between the mechanisms for controlling the exportation of components provided with these construct, and the restrictions placed on the use of collections as data objects.

In Figures 6.2 and 6.3 two approaches to the definition of a stack type in Euclid are shown; these examples are taken from [12]. The definition shown in Fig. 6.2 illustrates the basic characteristics of generalized records. The name 'stack' used in this example serves a function similar to that of 'tree' in the Simula example. It is bound to a template used to create instances of the 'stack' module. A particular instance of the 'stack' module would be bound to a variable defined by a declaration such as

```
var pd : stack(100)
```

```
TYPE stack(stacksize: unsignedInt) = MODULE
     EXPORTS(pop,push)
     VAR intstack: ARRAY 1..stacksize OF signedInt
     VAR stackptr: 0..stacksize := 0

     PROCEDURE push(x: signedInt) =
          IMPORTS(VAR intstack, VAR stackptr, stacksize)
          BEGIN
          PROCEDURE overflow = . . . END overflow
          IF stackptr = stacksize THEN
               overflow
          ELSE
               stackptr := stackptr+1
               intstack(stackptr) := x
          END IF
          END push

     PROCEDURE pop(VAR x: signedInt) =
          IMPORTS( VAR intstack, VAR stackptr)
          BEGIN
          PROCEDURE underflow = . . . END underflow
          IF stackptr = 0 THEN
               underflow
          ELSE
               x := intstack(stackptr)
               stackptr := stackptr-1
          END IF
          END pop

     END stack
```

Figure 6.2. A Euclid Stack Type Definition.

The components of such an instance can be accessed by selection as in the statement

```
pd.push(y)
```


The most significant difference between the generalized record proposals and the class is that not all the components need be accessible through selection. In the example, the variable components are not accessible because they are not included in the definition's export list. As a result, the definition completely hides the implementation of stacks from its users. In Ada, similar restrictions on the accessibility of components can be expressed by placing component declarations in the 'private' part of a package definition. In Mesa, the keywords public and private can be used to control accessibility. These extensions greatly increase the usefulness of the generalized records.

Another important difference is that these proposals allow types as components and extend controls of exportation to these components. To understand the importance of the existence of type components, consider the alternate approach to defining a stack type shown in Fig. 6.3. Unlike the preceding definition, this module has no variable components. Instead, it has a type named 'stk'. The elements of the type 'stk' are records that have as components the variables that were included in the earlier stack module. The procedures this new module take elements of the type 'stk' as parameters and access the components of these parameters as the procedures of the earlier definition accessed the variable components of 'stack' itself. The type name 'stk' is included in the module's export list, so

```
TYPE stack_type = MODULE
     EXPORTS (stk, pop, push)
     TYPE stk(stacksize: unsignedInt) = RECORD
          VAR stackptr: 0..stacksize := 0
          VAR body: ARRAY 1..stacksize OF signedInt
          END stk

     PROCEDURE push(VAR istk: stk(PARAMETER),
                  x: signedInt ) =
          BEGIN
          PROCEDURE overflow = . . . END overflow
          IF istk.stackptr = istk.stacksize THEN
               overflow
          ELSE
               istk.stackptr := istk.stackptr+1
               istk.body(istk.stackptr) := x
          END IF
          END push

     PROCEDURE pop (VAR istk: stk(PARAMETER),
                  VAR x: signedInt) =
          BEGIN
          PROCEDURE underflow = . . . END underflow
          IF istk.stackptr = 0 THEN
               underflow
          ELSE
               x := istk.body(istk.stackptr)
               istk.stackptr := istk.stackptr-1
          END IF
          END pop

     END stack_type
```

Figure 6.3. An Alternate Euclid Stack Type Definition.

code outside the module definition can use the name in declarations and parameter specifications. The export rules of Euclid, however, specify that any exported type is

'opaque'. That is, all the operations on elements of the type are hidden outside of the module, including operations of component selection. Thus, this alternate definition also hides the details of the implementation of stacks.

The difference between these two definitions is that in the first, the operations that manipulate elements of the type are components of the elements, while in the second they are collected into an independent type manager. This is precisely the difference between the kinds of types defined using the Monitor and those defined using the Cluster. Like Envi, the proposals for generalized records support both forms of type definition. Accordingly, they provide the facilities to define types with binary operations, as well as types like the stack.

The key to this is the ability to specify the details of a type manager explicitly, rather than relying completely on a language defined type manager which is little more than a template. Thus, in the definition of 'stack_type', the name 'stack_type' is itself bound to a language defined type manager. The objects produced using this type manager, however, also function as type managers. They provide for the creation of objects of the types they define through their 'stk' components, and they supply operations to apply to these objects. Similar type definitions can be constructed in the other generalized record languages, although their rules for the exportation of type components differ slightly.

Like Simula, the generalized record proposals fail to present as uniform a space of data objects as Envi. First, although they allow the programmer to define objects that behave as type managers, these type managers do not behave identically to those provided implicitly by the language. The name 'stack_type', which is bound to a language supplied type manager, can only be used as a type name in variable declarations and formal parameter specifications. On the other hand, if a name is bound to a user defined type manager through the declaration

```
VAR pd_type : stack_type
```

it can be used as an expression itself whose value is the type manager or in expressions that select one of its components, but nat as a type name. Similar statements apply to Ada package declarations.

In addition, generalized record proposals distinguish between simple types, records and procedures. Envi, on the other hand, views all these objects as special uses of its single class of data objects, environs.

A final important difference between Envi and these proposals involves the approach taken to encapsulation. Envi is based on the belief that compile time specification

of restrictions on access to objects belongs in the domain of the type system. Thus, in Envi, the facilities for specifying export limitations are not part of the syntax of the specification expression, which is an object creation mechanism. Instead, encapsulation is performed through appropriate type specifications in the definitions of functions that manipulate encapsulated objects. Recall that the Envi <u>module</u> construct is just a special syntax for accomplishing a common form of information hiding that can be performed in Envi without the <u>module</u>. The advantage of separating these facilities from the object creation construct is that it is easier to vary the encapsulation constraints associated with an object. In the disk scheduler example shown in Chapter 5, one object was encapsulated in four different ways in different contexts. This would be impossible in any of the generalized record proposals.

6.1.3. Synchronization Proposals

One of the first descendants of the class was the monitor proposal described by Brinch Hansen [8] and Hoare [31]. The monitor proposal was followed by many similar constructs, including the task of Ada [57], Distributed Processes [9], Synchronizing Resources [4], guardians [47]. These constructs all use the idea of collecting data values and procedures as components of one construct as the basis of a mechanism for synchronizing access to the data in the collection.

Although these proposals are similar from our viewpoint, they each provide a different set of names for the objects involved. So, we begin by establishing some common nomenclature. Each proposal provides a syntactic construct that we will call a _shared module specification_. These specifications include monitor declarations, Ada task type declarations, etc. Each shared module specification either directly produces a _shared module_ at run-time or produces a _shared module type_ consisting of shared modules. Shared modules are similar to objects of classes. They contain components including variables and procedures, where by procedures we mean to include procedure-like entities including Ada entries and SR operations. Control over parallel accesses to shared modules is provided through restrictions placed on parallel references to the procedural components of shared modules.

The most important differences between the synchronization proposals and the generalized record proposals result from the fact that the synchronization proposals have specialized the concepts introduced by Simula to the domain of parallel programming. For example, in order to ensure that interference does not occur, they do not allow the exportation of any variable components from shared modules. While

this may be reasonable in the context of concurrent programming, it limits the usefulness of these constructs in other contexts. To be more general, they should allow the programmer to export variable components when appropriate, as Envi and the generalized record proposals do.

A similar form of specialization can be seen in the mechanisms that the synchronization proposals provide for manipulating shared modules as data values. These proposals are not particularly concerned with this problem. They are satisfied to be able to access shared module through names to which they are statically bound. So, most of them do not provide features for assignment or parameter passing involving shared modules. In the original monitor proposal, in SR and with guardians, neither assignment nor parameter passing is supported. In Ada, parameter passing of elements of task types is allowed, but no assignment is provided.

While most of the synchronization proposals do not allow shared modules to be assigned or passes as parameters, those that do, do so in a manner very similar to Envi. This similarity derives from the fact that the modules involved are shared. In a conventional language, assignment involves the copying of values. With shared modules, this is undesirable because the goal is to provide another access path to a fixed shared module, not another module. Accordingly, while synchronization proposals do not provide conventional assignment operations for shared modules, some do allow assignment of pointers to shared modules. In Ada, this can be done by defining an access type based on a task type. The Dynamic Resource proposal of Andrews and McGraw [3] allows this type of assignment through capabilities that refer to shared modules.

The synchronization proposals also do not provide a data space as uniform as Envi's. The introduction of shared modules in synchronization proposals is generally accompanied by implicitly defined type managers. In Hoare's proposal, a monitor declaration can be turned into a monitor type declaration by adding the keyword class. Here, the name declared is bound to a type manager that can be used in other declarations to produce instances of the shared module desired. In Ada, the name of a task type is bound to a similar object. This name can be used in formal parameter specifications, as well as in declarations of instances of the type. While these type manager are simple, their presence requires that extra features be provided in each language mechanism that involves types. For example, in Ada, the rules for generic parameter matching require special provisions for task types.

In considering the synchronization proposals, we have thus far concentrated on their relationship to Envi, ignoring the differences between them. There is one important

area in which they differ from each other, however: the way in which procedural components are described. In the monitor proposal, the procedural components of shared modules are simply procedures. The rules describing the behavior of a monitor procedure are obtained by simply adding a mutual exclusion constraint to the normal procedure execution rules of the language. In later proposals, including Ada and SR, this has changed. The procedural components in Ada are called entries; in SR they are called operations. Entries and operations can be viewed as message channels. A call causes a message to be sent. The shared module to which the operation or entry belongs contains a process that receives messages sent to this procedural component. After a message is received and processed, a reply is sent to the caller, and both the caller's process and the process that processed the message proceed independently.

For the user of a shared module, the difference between the monitor view and the message passing approach is insignificant. On the other hand, the message passing approach gives the definer of the shared module the flexibility to specify more complex synchronization constraints easily, as well as the burden of specifying all synchronization constraints explicitly. Envi follows this second approach to the definition of procedural objects. Envi's operations are merely a slight modification of those found in SR.

The decision to use SR's operation mechanism in Envi was based on two factors. One is the increase in synchronization flexibility provided by the message passing approach. A second consideration was that SR'S shared module, the resource, suggested a natural way to incorporate processes into Envi. Since Envi was designed to be a language that combined facilities for synchronization and data abstraction, it was necessary to include processes in the language. If the processes were treated as objects independent of the environ, the language would include mechanisms for manipulating processes that would duplicate many of the mechanisms provided for environs. By modeling the environ after the SR resource, this was avoided. The process is a natural component of an object like the environ, and facilities for creating and manipulating environs provide all that is needed to manipulate processes.

6.1.4. Data Abstraction Proposals

The proposals that we have categorized as data abstraction proposals include Alphard [69], CLU [46], and Russell [17]. Like the synchronization proposals, these can be viewed as attempts to specialize the concepts of the class and other generalized records. The goals of the data abstraction proposals, however, are different from those of the synchronization proposals. The differences between the approaches taken to object definition by these two groups of

proposals have been discussed in Chapters 2 and 5. Here, the nature of the data objects of these languages is discussed.

Each of the data abstraction proposals supports two distinct sorts of aggregate objects. Type definitions made using the cluster of CLU, the from of Alphard or the type extension facilities of Russell all produce aggregate objects that act as type managers. The components of these type managers are procedures. The elements of the managed types are generally also aggregates, but they can only contain variables and constants, not procedures. Furthermore, as we will see, Alphard and CLU depend on a third class of objects that also act as type managers but are not aggregate objects at all.

As an illustration of these points, consider the CLU type definition shown in Fig. 6.4. This definition has been derived by modifying the tree type definition found in [45] to increase its similarity to the Simula tree definition given above. Nevertheless, the objects involved in the use of this definition are different from those in Simula.

In CLU, the name 'tree' is bound to a special type of function. The special characteristics of this function are that it can return types as values and take types as parameters. When one applies this object to some type, x, the result is a type manager for the subtype of trees of type x.

```
tree = CLUSTER[ t: TYPE] IS create, insert, find
  WHERE t HAS
      equal, lt: proctype( t, t) RETURNS( BOOL );

  node = RECORD[ value: t,  count: INT,
                 lesser: tree[t],
                 greater: tree[t

  REP = ONEOF[ empty: NULL, non_empty: node];

  create = PROC() RETURNS( CVT );
      RETURN( REP$make_empty(NIL));
    END create;

  insert = PROC( x: CVT, v: t) RETURNS (CVT);
    TAGCASE x
      TAG empty:
         n: node := node${value: v, count: 1,
                          lesser: tree[t]$create(),
                          greater: tree[t]$create() };
         RETURN( REP$make_non_empty (n));
      TAG non_empty ( n: node ):
         IF t$equal( v, n.value )
             THEN n.count := n.count + 1;
         ELSEIF t$lt( v, n.value)
             THEN n.lesser := tree[t]$insert(n.lesser,v);
         ELSE
             n.greater := tree[t]$insert(n.greater, v);
         END;
         RETURN( x );
      END;
    END insert;

  find = PROC( x: CVT, v: t) RETURNS (INT);
    TAGCASE x
      TAG empty:
        RETURN( 0 );
      TAG non_empty ( n: node ):
        IF t$equal( v, n.value )
            RETURN( n.count );
        ELSEIF t$lt( v, n.value )
            THEN RETURN (tree[t]$find( n.lesser, v));
        ELSE
            RETURN ( tree[t]$find( n.greater, v ) );
        END
      END
    END find;
END tree;
```

Figure 6.4. A CLU Tree Type Definition.

This type manager is an aggregate object formed from the components described in the cluster construct: the types 'node' and 'REP', and the procedures 'create', 'insert' and 'find'. The type components can not be accessed from outside the cluster. To actually produce an element of the type, one of the procedural components of the type manager must be invoked. In the tree example, and most other clusters, the 'create' operation is used.

The reader should recognize strong similarities between the two-level approach to type definition used in several of the Envi examples given in Chapter 5 and this CLU type definition. The difference is that CLU is specialized to support this particular approach to type definition, while in Envi it is just one of several approaches that can be used. In CLU, distinct sorts of objects are provided for each level in the definition, whereas the environ is used at all levels in Envi. In CLU, the function bound to the name 'tree' is a special object. Such functions can only be produced by cluster definitions. Special syntax is required for the where clause that describes its parameters. In a similar Envi definition, 'tree' would be the name of an environ with an operation named 'create'. The parameter and return value specifications for this operation would use the standard type specification mechanism of the language. The object returned by such an operation would simply be another environ that functions as a type manager, while the type managers produced by 'tree[x]' in CLU belong to a special class of objects used as type managers. Such object can only be produced by applications of the special functions described by cluster definitions. Finally, the objects produced by the 'create' operation of a 'tree' subtype type manager in CLU are different from the other sorts of CLU objects that have just been discussed. In this example, 'create' returns objects that are actually elements of the built-in structured type 'oneof'. In general, any built-in or user defined type can be used. Thus, the elements can be built up from simple types and the built-in structured types such as arrays, records, and unions. In Envi these objects would again simply be environs.

Similar specialization occurs in the other data abstraction languages. In Alphard, the <u>form</u> construct enables a programmer to associate a name with a type producing function in a manner similar to the use of a cluster. Applications of a form name to parameters can be used as type specifications in variable declarations that produce initialized variables of the type. The form appears to be somewhat different from the cluster in that it supports the declaration of both variable and procedural components. Unless the keyword <u>shared</u> is included, however, each declaration of a variable component in a form is actually treated as a specification of a selector function that can be applied to values of the type being defined to obtain a

reference to a variable of the component type. Thus, each variable of the defined type behaves as a record with components corresponding to the variable component declarations appearing in the form. In some sense it is as if each variable component declaration in a form declares two components. One is a variable that is a component of the objects used to represent the type's values. The other is a procedural component of the type manager that is used to select components of the objects that represent the type's values. Thus, the same distinction between aggregates of procedures and aggregates of values is made.

The Russell language does not make quite as many distinctions. If trees were defined in Russell, the name 'tree' would not have to be bound to some specially defined type producing function, because any Russell function can take types as parameters or return a type. The standard syntax for functions can be used with a 'type denotation' that describes the function's return value. The distinction between type modules and data values is still maintained in Russell, however. Type modules are not constructed from scratch as they are in CLU, Alphard and Envi. Instead, they are obtained from existing type managers by adding or hiding operation components. Thus, to define the tree type, one would use a built-in type constructor to obtain a type manager for the underlying record type, then modify this module by adding 'insert' and 'find' operations implemented in terms of the record type's selection operations, and finally hide the selection operations. Type managers are still aggregates of procedures while aggregates of values are treated separately. The differences between type signatures and value and variable signatures in Russell reflects this distinction.

This section, like the three that have preceded it, has emphasized the fact that Envi, unlike the other languages discussed, does not distinguish between type managers and data values. The advantages of Envi's approach have also been discussed. It provides considerable programming power with a minimum of mechanism. The motivations that led to these distinctions in other languages have not, however, been presented.

There are arguments based on efficiency and clarity that support the approaches used in other languages. The use of special constructs for the definition of types and shared objects, together with the use of distinct objects to support these types enables the compiler and the reader of a program to quickly determine a good deal of information about the ways in which certain objects in the program will be used. In fact, the ability to distinguish type managers from other objects immediately in these languages is more than useful -- it is necessary. The type systems of these languages depend on it. The ability to describe a type in a

parameter specification by merely naming an object depends upon the ability to determine that the object named is a type manager. To determine this, a language must have a fixed notion of which objects are type managers. Thus, Envi's uniform data space is only possible because of its unusual type system.

6.2. Expressions

Envi's treatment of expressions is probably the most novel feature of the language. First, through the specification expression, Envi provides an unusual approach to the creation of new objects. Second, expressions are used as type specifications. In this section, these aspects of Envi's expressions are compared to other languages that provide similar facilities.

6.2.1. Object Creation

In most languages, the creation of new objects is associated with the mechanisms for procedure invocation, or with the language's type system, or both. Many Algol-like languages provide for dynamic allocation of variables and arrays during procedure calls and block entries. Other facilities for more dynamic allocation, such as Pascal's heap, generally involve the definition of a "dynamic" type to which a "new" operation can be applied to produce new objects. Envi differs from such proposals in that it provides a single mechanism for creating new objects that is independent of both the type system and the procedural abstraction facilities. In Envi, new objects are produced solely by the evaluation of specification expressions.

6.2.1.1. Object Creation and Typing

The facilities provided in Envi by the specification expression are most closely related to mechanisms such as Pascal's heap, Ada's Access types, Simula's new operation and the type constructors of CLU. Each of these mechanisms enables a programmer to produce a new object by evaluating an expression. In each language, the result of this expression evaluation is a pointer. In Pascal, Ada and Simula, this is made explicit in the language. In Envi and CLU, the use of pointers is not explicit, but the semantics of assignment require the sharing of mutable (i.e. changeable) objects, thus suggesting the use of pointers in the implementation. All these mechanisms imply the use of dynamic storage allocation algorithms to manage the space needed for created objects.

The difference between the other mechanisms and Envi's is that the expressions that are used to produce new objects in these other languages all do so by referring to the type to which the produced objects will belong. For example, in Simula, we have seen that the name of the class to which the new object will belong is used in an expression, such as


```
NEW tree(x)
```

Similar notations involving type names are used by Ada and Pascal. This is not the case in Envi. The specification expression enables one directly to describe the objects to be produced in the expressions that produce them.

One of the justifications for this approach is philosophical. We accept as a basic design principle that any syntactic construct that describes an object should produce what it describes. Surprisingly, the approaches to object creation found in many languages violate this simple principle. In Simula, each class declaration gives a clear description of the objects desired. Elaboration of such a construct, however, does not produce the object described. Instead, it produces a type manager that can be accessed through the class name and used to produced the objects described.

Envi attempts to avoid this separation. The environ specification expression is syntactically similar to the class. In Envi, however, elaboration of this construct actually produces the object described. There is no need to apply any 'new' or 'create' operation to the object produced.

The fact that Envi does not base its object creation mechanisms on type managers, however, does not mean that type managers are not used for object creation. In most of the examples presented in Chapter 5, a type manager was first produced and then new objects were obtained by calling its 'create' operation. This is true of all the queue definitions and of the definition of the rational type. It is not surprising that Envi uses this technique. If it were not a useful technique it would not have led to the introduction of specialized constructs to support it in so many languages. The difference is that Envi supports this technique through the specification expression rather than through specialized constructs. This approach has several consequences.

First, Envi's use of the specification expression is essential to the uniformity of its data objects. In comparing the environ to the data objects in other languages, we noted that in many languages with facilities for data abstraction, the type managers associated with user defined types form an extra class of data values. In Envi, the fact that any environ can contain an operation that creates new objects using specification expressions makes it possible to use an environ as a type manager. Thus, one obstacle to the uniform treatment of data objects is removed.

Another consequence of the use of the specification expression is increased flexibility in the way types and new objects can be produced. Of particular interest is the

ability to create objects without associating them with a type. In its simplest form, this flexibility simplifies the programmer's task by enabling one to create objects without defining an appropriate type when such a type definition would be superfluous. For example, in the solutions to the readers/writers problem shown in Chapter 5, the servant objects that processed user requests were produced by in-line specification expressions.

Another benefit of this flexibility is that it enables one to postpone the association of a type manager with an object. The precise control over access rights to the servant objects used in the solution to the disk scheduling problem discussed in Chapter 5 illustrates this. The objects originally produced by invoking the 'rserv' and 'wserv' operations are not associated with a type manager until just before they are passed to the queue management primitive 'insert'. This enables code in the body of the scheduler to access these objects as untyped environs, yet still places access restrictions on the queue management routines through the type manager 'dserv'.

As one might expect, the flexibility provided by the specification expression makes it somewhat more difficult to give simple type definitions. To provide more options to the programmer, Envi must require that the programmer supply more details in such specifications. One can see the extent to which this can occur by comparing the node type definition used in the mergeable queue defined in Chapter 5 with the roughly equivalent, Pascal definition shown in Fig. 6.5. The length of Envi's definitions suggests that abbreviations for them might need to be added to the language, as they were for other common constructions.

6.2.1.2. Object Creation and Invocation

Frequently, the invocation of a procedural abstraction should be accompanied by the allocation of a new environment in which the computation invoked can be performed. Nevertheless, allocation and invocation are separate programming activities. Accordingly, in Envi, they are treated as such. In this section we will compare Envi's approach with conventional procedure mechanisms and other proposals that separate invocation and allocation including SL5 [24,28] and Ada [57].

```
TYPE ref  = ↑node;
     node = RECORD
              data: val;
              is_last: Boolean;
              next: ref
            END
```

Figure 6.5. A Pascal Record Type.

The invocation of a procedure in an ALGOL-like language is a complex action that can be decomposed into at least three identifiable subparts: allocation, data transfer and control transfer. While these three parts exist in most languages, they are generally inseparable. An important exception to this is SL5. In SL5, the three actions involved in conventional procedure call can be performed independently. We will begin by examining this language in order to clarify the nature of the three aspects of invocations and to study the advantages and disadvantages of separating them.

In SL5, procedures are data values that are produced by procedure descriptions and stored in variables through assignment. Thus, the assignment shown in Fig. 6.6 stores a procedure that computes the greater common divisor of its parameters into the variable 'gcd'. In most languages, a

```
gcd := PROCEDURE (x,y)
           WHILE x ~= y DO
              IF x > y THEN x := x - y
                       ELSE y := y - x
           SUCCEED x
        END
```

Figure 6.6. An SL5 Procedure Definition.

procedure value is accessed in call statements or function calls. In SL5, a procedure cannot be called. Instead, procedure values are used in _create_ expressions which return objects called _environments_. It is these environments that are actually called, not the procedures.

The process of calling an environment is further divided into binding and resuming. Actual parameter values can be passed to an environment by placing the argument expressions in a _with_ expression. For example, if an environment 'e' is produced from the procedure assigned to 'gcd' through the expression

```
e := CREATE gcd
```

then the values of 'a' and 'b' can be passed to 'e' through the evaluation of the expression

```
e WITH (a,b)
```

A _with_ expression returns the environment as its value. Finally, control is transferred to the environment through a 'resume' expression. Thus,

```
RESUME e
```

would transfer control to the gcd procedure. The value of

the _resume_ expression is the value returned by the resumed environment.*

The three SL5 operations, _create_, _with_, and _resume_ correspond precisely to the three aspects of conventional procedure mechanisms: allocation, data transfer and control transfer. In fact, SL5 provides for conventional function calls by viewing them as abbreviations for references to its more primitive facilities. Thus, the expression

```
gcd (a,b)
```

is allowed in SL5 as an abbreviation for

```
RESUME (CREATE gcd WITH (a,b))
```

Envi falls between SL5 and conventional procedure mechanisms. In Envi, the invocation of an operation involves both data transfer and control transfer, but not allocation. Allocation is performed using the specification expression, independently of the operation mechanism. Thus, if a 'gcd' function similar to that shown in SL5 were desired, the first step would be the evaluation of the expression shown in Fig. 6.7. The object produced corresponds to the environment of SL5, not the procedure.

* In SL5, procedures return both a value and a signal indicating success or failure. Thus, the _succeed_ statement is just a return statement that returns the signal for success with the return value given.

```
ENVIRON
     (gcd: OP (x,y: integer): integer;
      xt, yt: var (integer);
      SELECT
          gcd (x,y) -->
               xt:= x;
               yt:= y;
               DO xt > yt -->  xt := xt - yt
                  yt > xt -->  yt := yt - xt
               OD;
               RETURN(xt)
      END
     )
```

Figure 6.7. An Envi Gcd Procedure.

That is, it is used to compute greatest common divisors, rather than to produce objects that do this.

If the object produced by the specification expression in Fig. 6.7 is bound to the name 'e', it can be used to compute the gcd of 'a' and 'b' by evaluating the expression.

```
e $ gcd (a,b)
```

Again, this is different from SL5. The transfer of the values of 'a' and 'b' is performed together with the transfer of control to the code that determines the gcd. The separation of SL5's with and resume operations is not provided.

Despite the differences noted between Envi and SL5, the two languages' invocation mechanisms share most of the same advantages and disadvantages when compared to conventional approaches. So, we will first consider their common features. Then we will discuss the differences between Envi and SL5.

The advantage of separating the allocation of environments from the transfer of parameters and control is largely a matter of flexibility. The programmer can precisely tailor the details of a particular procedural abstraction to the requirements of the context in which it is used. The disadvantages arise from the same factors. Since the programmer has more flexibility, more of the details of procedure definition must be made explicit and the implementation must provide support for more alternatives. The balance between the advantages and disadvantages is not, however, fixed. It depends on the nature of the procedural abstractions involved. The disadvantages outweigh the advantages when simple recursive procedures are involved, but just the opposite is true when coroutines are considered.

The only type of procedure that requires the allocation of a new environment for each invocation is the recursive procedure. With such a procedure there is no advantage in being able to separately specify environment allocation and invocation. Doing so merely increases the syntactic complexity and decreases the compilers ability to use efficient stack based allocation schemes for environments. These disadvantages are reduced in SL5, and to a lesser degree in Envi, by the introduction of the standard procedure call syntax as an abbreviation for the separate specification of allocation and invocation (see Section 4.1.3.2).

Another class of procedures that benefit from the association of allocation with invocation in conventional languages are polymorphic procedures. If a procedure is polymorphic, some details of the environment in which its execution is performed may depend on its parameters. In this case, a new environment may need to be allocated for each set of parameters. An example of this sort of allocation was shown in the polymorphic polynomial evaluator in Chapter 5. Each invocation of the evaluator was actually split into stages. First an environment containing a temporary variable of the appropriate type was allocated. Then, the necessary computation was performed in this environment. Much simpler forms of polymorphism also require such allocation. In Algol procedures, the shapes of arrays declared locally may depend on parameter values. As a result, the environments containing these arrays must be allocated after data transfer.

Unlike recursive procedures, polymorphic procedures do not require environment allocation with each invocation. When the number of distinct types that the procedure must handle is limited, the separation of allocation from invocation makes it possible to pre-allocate all the needed environments. This sort of pre-allocation is possible in Envi. A user might retrieve the definition of the polymorphic polynomial evaluator knowing that it would only be used for 'real' and 'complex' polynomials. The definition of 'polyval' provides for this possibility through the 'create' operation. The user can define two objects through the declarations.

```
realpoly:  polyval $ create (real)
compoly :  polyval $ create (complex)
```

These objects can then be used to evaluate polynomials without further allocation. This would be accomplish by expressions such as

```
realpoly $ eval (coefs, degree, value)
```

This facility enables the programmer to eliminate some of the inefficiencies associated with the powerful polymorphism facilities of a language like Envi.

A final factor that requires the allocation of multiple instances of a procedure's environment is the presence of parallelism. It is necessary to allocate at least one instance of a procedure's environment for each process that calls it. In systems in which the number of processes varies, this allocation must be performed dynamically. As with polymorphism, the programmer can reduce the overhead involved by explicitly allocating an environment for each process that uses the procedure so long as mechanisms are provided to separate allocation from invocation. Moreover, in applications where parallelism is present, such explicit pre-allocation often fits naturally into the specification of an abstraction.

Consider the solutions given to the readers/writers problem in Chapter 5. In all these solutions, the specification expression was used to mimic the behavior of a conventional procedure mechanism. A new environ was created on each invocation of 'read' or 'write'. These objects are not all needed. If a separate operation that produces environments through which the data base can be accessed is provided, only one allocation is required for each process. For example, in the implementation of data bases, the 'open' operation provides a natural place to handle allocation. Fig. 6.8 shows a readers/writers data base implementation in which the allocation of environments is performed as part of the 'open' operation. The 'open' operation first checks the caller's authority to access the data and then creates and returns an environment that processes 'read' and 'write' requests. (No corresponding 'close' operation is provided


```
MODULE rw IS
   open
FROM
  ENVIRON RW
    ( open: OP( id: password ):
                    OBJECT WITH
                       ( read: OP( i: integer ): item;
                         write: OP( v: item; i: integer)
                       );
      startread, startwrite, endread, endwrite  : OP();

      state, : VAR( integer INIT 0);
      database : array$create(item,integer)$create(1,n);

      SELECT open( id ) -->

                 < validate authority to access data >

              RETURN( ENVIRON
                        ( read : OP( i: integer) : item;
                          write: OP( v: item; i: integer);

                          output : VAR( item );

                          SELECT read( i ) -->
                             startread();
                             output := database[i];
                             endread();
                             RETURN( output)

                          [] write( v, i ) -->
                             startwrite();
                             database[i] := v;
                             endwrite();

                          END )
                         )

      [] startread() WHEN state => 0   -->
              state := state + 1;

      [] endread() --> state := state - 1;

      [] startwrite() WHEN state = 0    -->
              state := -1

      [] endwrite() --> state := 0
      END )
```

Figure 6.8. Readers/writers with an Open Operation.

because it would serve no function is this example.)

In parallel programming, the ability to separate allocation from invocation provides advantages in addition to the ability to reduce allocation overhead. When environments are allocated on a per process basis, information about the processes can be stored in the environments. For example, in the readers/writers solution, the 'open' operation could store the fact that a user's access should be read-only within the created environment. This enables the write operation to reject illegal invocations without first accessing a global list of access rights. Thus, the environments allocated can function somewhat like capabilities [18,40].

The ability to reuse an environment of execution provides SL5 and Envi with capabilities beyond those of conventional mechanisms. The fact that access rights can be stored in an environment is an example of such a capability. More interesting examples arise when the data saved in an environment is changed by each invocation or includes the instruction pointer. Such environments enable Envi and SL5 to support coroutines.

While coroutines have been recognized as a useful control abstraction for some time [14,52], they are poorly supported by conventional languages. Mechanisms that separate allocation from invocation provide better support for these

constructs. An example of the use of coroutine structures in Envi is shown in Fig. 6.9. This example is yet another version of the readers/writers problem. In this version, a more accurate model of a data base is used. The users are assumed to need access to more than one record during each transaction. Thus, a read transaction may involve several actual reads and a write transaction may involve several reads and writes. The solution supports this by viewing the transaction as an ongoing computation that proceeds in segments each time that it is resumed by the user. It begins when the user gives the routine control by invoking the 'readhold' or 'writehold' operation. The process can then service any number of read and write requests until a 'free' request causes the computation to end.

Before leaving the subject of invocation, the difference between Envi and SL5 should be considered. The important differences between Envi and SL5 are similar to the difference between Envi and the dynamic allocation mechanisms discussed in the preceding section. In SL5, the procedure definition construct describes an environment. Elaboration of such a construct, however does not produce an environment. Instead, evaluation of a procedure definition produces a 'procedure', which can later be used to produce the environments described. This approach involves the same inconsistency that was found in dynamic type facilities -- the procedure's definition doesn't directly produce the

```
MODULE rw IS open
FROM
  ENVIRON RW
   ( open: OP( id: password ):
                OBJECT WITH
                   ( readhold, writehold, free : OP();
                     read: OP( i: integer ): item;
                     write: OP( v: item; i: integer )  );
     startread, startwrite, endread, endwrite  : OP();
     state, : VAR( integer INIT 0);
     database : array$create(item,integer)$create(1,n);

     SELECT open( id ) -->. . .<check access rights>. . .
       RETURN(
          ENVIRON
           ( readhold, writehold, free: OP();
             read : OP( i: integer) : item;
             write: OP( v: item; i: integer);
             writer, done: VAR( boolean INIT false );

             PROCESS
               DO true -->
                  writer := false; done := false;
                  IN readhold(  ) --> startread()
                  [] writehold( ) --> startwrite();
                                      writer := true
                  NI;
                  DO ~ done -->
                     IN read( i ) -->
                          RETURN( database[i] )
                     [] write(v,i) WHEN writer  -->
                          database[i] := v
                     [] free() --> done := true;
                     NI
                  OD;
                  IF ~ writer --> endread()
                  [] writer --> endwrite()
                  FI
                OD
              END   )  )
     [] startread() WHEN state => 0   -->
              state := state + 1
     [] endread() -->
              state := state - 1
     [] startwrite() WHEN state = 0    -->
              state := -1
     [] endwrite() --> state := 0
     END )
```

Figure 6.9. Readers/writers with an Open Operation.

objects that it describes. Also, it further complicates the data space of the language. SL5 requires distinct mechanisms for manipulating procedures and environments. In Envi, objects corresponding to both of these constructs are represented by environs, and are produced directly by specification expressions.

Several other languages provide invocation facilities that can be used independently of environment allocation. Most closely related to Envi is Ada, which accomplishes this separation by basing invocation on a message-passing model. In Ada, environments correspond to tasks. The allocation of new environments is therefore accomplished by allocating new tasks, which can be done with task types. This mechanism is logically similar to SL5's mechanism. The type managers used for task types correspond to SL5's procedures. They form a distinct group of objects and are produced by a syntactic construct that does not explicitly describe them.

6.2.2. Expressions as Type Specifications

It is not clear that anyone could give a definition of type that would be generally acceptable to all involved in the design and use of programming languages. Certainly, many have tried. Some believe that types are sets of data values [30], others that they are sets of operations [17] and still others think they are both. The safest approach seems to be to avoid taking a position, and that is what

Envi attempts to do. This statement may seen ridiculous in light of the fact that several Envi type definitions were exhibited in Chapter 5. It is, nevertheless, true.

The resolution of this apparent contradiction lies in a rather fine distinction. The fact that a concept can be used in a language does not imply that the notion is part of the language's definition. The special purpose language, ALTRAN [27], includes polynomials as a built-in data type. In general purpose languages, including Pascal, Algol and FORTRAN, it is possible to represent and use polynomials, but this does not imply that polynomials are a part of these languages' definitions in the same sense that they are part of ALTRAN's. Similarly, most languages provide special constructs for defining or using types. Envi allows the programmer to represent and use types, but this does not mean that types are incorporated into Envi any more than polynomials are into Pascal. In languages like CLU and Alphard, a type is what you get when you use the 'cluster' or 'form' constructs. In Envi, 'type' is a concept in the programmer's mind that can be expressed using environs. The appearance of a 'cluster' or 'form' always implies the definition of a type, while an environ may or may not define a type. In some cases, programmers might not agree whether a particular environ did or did not define a type.

These facts provide guidance to the study of Envi's features. They suggest that one should attempt to identify and examine those features of the language that, while not specifically tailored to type definition, provide programmers with the ability to use the notion of types. Two of these features have already been discussed. First, the data domain of the language has been designed to include objects similar in structure to objects that can only be produced by type defining constructs in other languages. Secondly, the environ specification expression provides the programmer with the ability to create new objects -- a capability that is often limited to the mechanisms of a language's type system. This section examines the third feature that enables Envi to support typing without including a fixed type mechanism: the technique of using expressions to place static constraints on the use of data values. This will be done by comparing Envi's features to the the type checking systems of Alphard and Russell. These languages are good examples because the capabilities they provide are similar to Envi's and their rules for type checking are clearly defined. We will concentrate on three features of each language: the syntax for type specifications; the rules for associating type specifications with expressions; and the rules for determining the compatibility of two type specifications.

In Alphard, the syntax for type specifications is divided into three parts. Nominal types are the specifications used to describe sets of objects of abstract types. A nominal type consists of a <u>base type</u>, which is any name associated with a form and a possibly empty set of <u>type qualifiers</u>. Examples include

    integer

and

    vector(real, 1, 10)

The base types in these examples are 'integer' and 'vector'. 'Real', '1', and '10' are qualifiers. If the expected parameter is a procedure, a <u>routine description</u> is used instead of a nominal type. A routine description is essentially a procedure heading. For example,

    proc(var a : integer)

describes a procedure with one parameter. Finally, if the objects involved are types, the syntax used to define types, the <u>form</u>, is used as a type specification syntax. The body of a form with the desired structure but without implementation details is used.

The rules for associating type specifications with Alphard expressions are given with the descriptions of the expressions. For example, if a function call is

encountered, its type is determined by elaborating the return value type specification given in the declaration of the function name used.

Finally, the rules for determining the compatibility of type specifications are divided into three parts corresponding to the three types of specifications supported. An actual parameter expression with nominal type 'T_A' is acceptable as a parameter with type specification 'T_F' only if T_F is a nominal type and T_F <u>subsumes</u> T_A. The definition of subsumes requires that T_A and T_F have the same base type and that the qualifiers of T_A are compatible with those of T_F. A procedural parameter is acceptable only if its routine description is identical to that given as a formal parameter specification, up to substitution of local names. Two form descriptions are compatible only if the actual parameter type description <u>syntactically satisfies</u> the formal parameter description. The definition of syntactically satisfies requires that the two types provide similar operations. The details of subsumption and syntactic satisfaction are not important here. Our goal is merely to identify the basic components of type checking systems, not to study the details.

The same components can be found in Russell. The syntax for type specifications is called the signature. The rules for determining compatibility are called the signature calculus, and rules for associating signatures with expressions are given with the definitions of the expressions. Russell's signatures are also divided into three groups. Values and variables are described by placing a type-producing expression after the keyword var or val. Thus the signatures,

    val integer

and

    var array[10, integer]

are similar to the Alphard nominal types shown above. These notations are not identical, however, since Russell allows a wide variety of expressions to describe types. Procedural values are described by signatures that resemble procedure headings. The Russell signature

    func[val : integer] val integer

is a simple example. Finally, types are described by the keyword type followed by a list of the descriptions of the operations that the type provides.

The rules of the signature calculus that determine type compatibility are simple. 'Val' and 'var' signatures must be identical. Procedure signatures must be identical up to the choice of local names. Type signatures match if the actual's signature can be made identical to the formal's

signature by changing local names, and reordering or deleting operation descriptions from the actual's signature.

Initially, Envi's type checking system appears more complicated than others. The only true test of this question is experience, but we believe that examination of the structure of Envi's system suggests that it will actually prove simpler than other systems that provide the same capablilities. Envi's system has fewer parts, and the parts that exist have a reasonable intuitive foundation.

The Envi system does not have the three components that Russell and Alphard have. In Envi, there is no distinct syntax for type specifications that the programmer must learn. Instead of including distinct notations for expressions and type specifications, Envi extends the syntax of expressions slightly to provide the ability to use them as type specifications. The use of the '...' in invocations provides the extra power needed. Therefore, the rules for associating type specifications with expressions can be removed. The only component left is the set of rules for determining type compatibility, and these refer to pairs of expressions, rather than to pairs of type specifications.

The rules for determining type compatibility are similar to those found in Russell and Alphard, but because they are associated with expressions rather than type specifications, a simpler intuitive basis can be given for them.

Chapter 3 explains that the type compatibility rules depend on an ordering on the set of expressions. The actual rules defining this ordering are fairly complex formalizations of the same basic transformations used in Russell and Alphard: substitutions for local names, and elimination or reordering of components. Each rule, however, can be understood in terms of the intuitive statement that if a $\geq$ b is true it will be valid to use 'a' anywhere that 'b' can be validly used.

The final aspect of the type system that provides a simplification is the elimination of distinct type specification notations for values, procedures and types. This simplification is in some sense unavoidable; since Envi only supports one sort of data value it only needs one sort of type specification. Be aware, however, that this special type system makes Envi's uniform data space possible by supporting data types without requiring special objects to represent types.

While the relative complexity of Envi's type checking system is debatable, there are other advantages that are more obvious. These advantages are due to the flexibility Envi gives to the programmer by not imposing a fixed notion of types.

One well known issue in type checking systems is the choice between structural equivalence and name equivalence

[65]. A language with fixed features for defining and using types is forced to choose one of these approaches. Envi, on the other hand, gives the programmer the flexibility to choose the approach most appropriate to each application. For example, a record type definition is shown in Fig. 10. A procedure can be written that will only use records produced by this type manager by using the type specification

```
FROM[ records$create(...) ]
```

for the parameter. If instead, a procedure that accepts any object whose structure is similar to those produced by 'records', the type specification

```
FROM[ ENVIRON( a: VAR(integer); b: VAR(boolean)) ]
```

can be used. The decision is made by the programmer, not by the language.

```
ENVIRON records
 ( create : OP(): FROM[ ENVIRON( a: VAR( integer );
                                 b: VAR( boolean ))
                      ]
   SELECT create() -->
      RETURN(
        ENVIRON( a: VAR( integer );
                 b: VAR( boolean ) )
             )
   END )
```

<u>Figure 6.10</u>. <u>An Envi Record Type Definition</u>.

Similar flexibility is available in the definition of subranges. The facilities incorporated in Envi for the definition of subranges provide subrange types whose elements are also recognized as elements of the base type from which they were extracted. If this is inappropriate, however, a programmer can easily define special subrange types whose elements cannot be used as members of the base type. The techniques for doing this were discussed with the descriptions of subrange types in Chapter 4.

The flexibility of this system applies to more than just simple data objects. In Envi, more complex objects, including type managers themselves, are subject to the same type system. If one examines the type specification mechanisms in most type abstraction languages, it becomes clear that restrictions placed on type parameters to generic definitions can only be based on structural similarity. In Envi, the programmer can choose to use either structural or name equivalence for these objects. The advantages of using name equivalence may be considerable. The restrictions used in most languages only ensure that type actual parameters have certain operations. They do not ensure anything about the semantics of these operations. By limiting the types accepted to those produced by a certain module, more stringent restrictions can be enforced.

6.3. Summary

In this chapter, the features that make the Envi language unusual have been examined in detail. This detail, however, may have obscured a basic fact about the language's innovations. The uniform data space and the unusual uses of expressions in Envi are all consequences of the belief that aggregate data objects like the environ are central to programming and that such objects should be treated as the elements underlying a language's data space, rather than as special constructs whose applicability is limited to data type definition, synchronization, separate compilation or any other single concern.

Envi's unusual data space supports this viewpoint by treating the environ as the only kind of data value. All the data manipulation mechanisms of the language are organized around these objects. The language avoids treating them or any other set of data values as special cases.

The specification expression and the expression-based type checking system provide the programmer with the ability to adapt the language's general purpose data objects to fit the special purposes for which these objects are often used. They enable the programmer to define data types without introducing a distinct sort of type managers to the data space. They provide for encapsulation of data type and shared object definitions. Furthermore, they enable basic data types such as the arithmetic types to be represented as elements of the data space built of environs.

Thus, the unusual features of Envi are not just additions to meet special programming needs. They are basic elements of a system that supports a new approach to data object management.

Chapter 7
Conclusions

7.1. Summary

In this thesis we have proposed new programming language features that address the problems of both parallel programming and data abstraction. We began by studying the relationship between these two problem areas. In Chapter 2, we showed that existing data abstraction and parallel programming proposals involve the use of the same fundamental mechanisms -- object creation, static access limitation and dynamic access limitation -- but that these mechanisms are used in different ways in each area. We argued that this created a serious problem. As a result of the differences in the uses of the fundamental mechanisms, any programming construct that combined the mechanisms in a way that was appropriate for one area would be unsatisfactory for the other. At the same time, because the fundamental mechanisms are the same, any language that provides distinct constructs for each of the problem areas is certain to contain redundant features. We concluded that the best approach is to provide the fundamental mechanisms through separate language features that the programmer can combine in various ways, rather than to define language constructs that support particular combinations.

In Chapters 3 and 4, we presented a language called Envi that provides the three needed mechanisms through three separate features. For object creation, the specification expression is included. This construct simply enables the programmer to create a new object by describing its components. It does not incorporate special features to support data encapsulation or to distinguish synchronized objects from unsynchronized objects. The mechanisms needed are provided independently by the languages type system and the operation construct.

ıhe type system we proposed is essential to the separation of object creation and static access limitation in the language. Unlike most type systems, it supports a method of type specification that does not depend on names defined using special type definition constructs. Instead, its type specifications are based directly on the expressions used in a program. Nevertheless, it provides all the facilities of other, sophisticated type systems.

Finally, to provide for the enforcement of dynamic access constraints required by synchronization problems, we incorporated Andrews' [4] operation mechanism.

In Chapter 5, we demonstrated that the language provided flexible tools for parallel programming and data

abstraction. We first showed that our constructs could be used to produce solutions to several standard synchronization and data abstraction problems. Then, we exhibited their special capabilities by considering problems which involved both parallelism and data abstraction and posed unusual encapsulation problems.

Finally, in Chapter 6, we compared the features proposed to those of other languages to clarify and justify our innovations.

7.2. Further Research

Much work remains to be done if a language like Envi is ever to serve as a practical programming tool. This thesis has shown that treating object creation, static access limitation and dynamic access limitation as primitive mechanisms is a promising approach to programming language design. As Hoare said of his Communicating Sequential Processes proposal, however:

> "It would be unjustified to conclude that these primitives can wholly replace the other concepts in a programming language. Where a more elaborate construction (such as a procedure or a monitor) is frequently useful, has properties which are more simply provable, and can also be implemented more efficiently than the general case, there is a strong reason for including in a programming language a special notation for that construction. The fact that the construction can be defined in terms of simpler underlying primitives is a useful guarantee that its inclusion is logically consistent with the remainder of the language. [32]"

This was the approach taken in extending Envi-0 to form Envi. But in this case, the choice of features to add was guided by educated guesses. To produce a practical programming language based on Envi-0 a more careful study must be made to determine precisely which constructions are "frequently useful", can "be implemented more efficiently than the general case" and have "properties which are more simply provable".

To identify constructs of the first two sorts, the language must be implemented. A language as different from others as Envi leads its users to develop new programming techniques. One cannot accurately determine which special features will be frequently useful until these techniques have been developed. Without any implementation of the language, it is unlikely that enough programming can be done to provide the needed information.

The implementation provided to support this initial experimentation with the language should be easily modifiable to accommodate changes made to the language as a consequence of the information obtained. Therefore, it should be primarily interpretive, which means that the efficiency with which Envi programs execute will be poor. Nevertheless, it is hoped that it will provide enough experience with the implementation of Envi's features to enable one to recognized additions to the language that "could be implemented

more efficiently that the general case."

While extensions made to the language can improve the efficiency of certain programs, ultimately, a sophisticated translator will be required if the most general features of Envi are to be implemented with reasonable efficiency. Many of the language's features are difficult to implement efficiently. For example, because of the object sharing permitted by Envi's assignment operation, a general memory management scheme involving garbage collection or reference counting must be used. The overhead involved in queueing messages and switching processes to implement operations is likely to be greater than that of a conventional procedure call. In addition, if an object is referenced through several names associated with different type restrictions, templates must be used to map the structure of the actual object onto the structures described by the type restrictors. The indirection these templates introduce will slow references to objects; the overhead involved in creating these templates will slow the assignment operation, and their presence will complicate memory management.

Despite these difficulties, we believe that Envi can be implemented efficiently. The techniques required will involve more than those used for simpler languages [29,68]. In particular, they will include techniques used for memory management [36,43] and type inference [37,59] developed for very high level languages like SETL [38] and Smalltalk [35]. For example, the problems associated with templates discussed above could be reduced if the compiler could deduce information about typing omitted in the program's type restrictions. We feel that these techniques may be more successfully applied to Envi than to the languages for which they have been developed because in Envi the programmer provides more static information about the program.

In addition to these practical problems that must be addressed if Envi is to become the basis for a usable language, there are some theoretical issues raised by the language. It would certainly be desirable to have a formal definition of the semantics of the language in either an axiomatic or a denotational formalism. Unfortunately, the language's features for parallelism make a denotational definition difficult, while the presence of side-effects inherent in the behavior of environs would complicate an axiomatic definition. While some work has been done on the formal definition of such languages, the difficulties involved may make it more appropriate to study properties of interesting sub-languages of Envi.

One interesting question that might be studied in this way is the nature of Envi's type system. In the design of Envi, it became obvious that languages with complicated rules for type correctness require some means to determine

that the rules indeed formalize the desired intuitive notion of type correctness. This could be done for Envi by considering a sub-language in which processes and operations are replaced by a conventional procedure mechanism, thus eliminating parallelism. Such a language could be described using the methods of denotational semantics. This would make it feasible to show that Envi's type system was correct in that it guaranteed representation independence.

Appendix A. Summary of ENVI-0 Syntax

```
< expr > ::= ENVIRON ( < id > } ( < comp spec >;* )
          |  < id >
          |  < expr > $ < id >
          |  < expr > $ < id > ( < expr >;* )
          |  < expr > $ < id > ( ... )
< comp spec > ::=  < const decl >
               |   < var decl >
               |   < op decl >
               |   < process >
< const decl > ::= < id >,+ : < expr >
< var decl > ::=
     < id >,+ : VAR( < type rest > {INIT < expr > } )
< type rest > ::= FROM[ < expr > ]
< op decl > ::= < id >,+ : < op desc >
< op desc > ::= OP( < parm spec >;* ) {: < type rest > }
             |  OP( ... ) : < type rest >
< parm spec > ::= { < id >,+ : } < type rest >
< process > ::= PROCESS < stmt >;+ END
< stmt > ::= SKIP
          |  < var > := < expr >
          |  < expr > $ < id > ( < expr >,* )
          |  IF < guarded command > [] + FI
          |  DO < guarded command > [] + OD
          |  IN < input command > []+ NI
          |  RETURN( < expr > )
```



```
< var > ::= < expr > $ < id >
< guarded command > ::= < expr > ---> < stmt >;+
< input command > ::=
    < id >( < id >,* ){WHEN < expr >} ---> < stmt >;+
```

Appendix B. Summary of Type System Closure Rules

Number	Description	Page	Section
Rule TR.	Transitivity	69	3.3.1.2.2
Rule RF.	Reflexivity	69	3.3.1.2.2
Rule SS1.	Component Deletion	70	3.3.1.2.2
Rule SS2.	Component Reordering	72	3.3.1.2.2
Rule SS3.	Environ Renaming	73	3.3.1.2.2
Rule SS4.	Constant Compatability	73	3.3.1.2.2
Rule SS5.	Variable Type Compatability	74	3.3.1.2.2
Rule SS6.	Variable Initialization Hiding	75	3.3.1.2.2
Rule SS7.	Operation Type Compatability	75	3.3.1.2.2
Rule SS8.	Operation Parameter Renaming	76	3.3.1.2.2
Rule SS9.	Operation Parameterization Hiding	76	3.3.1.2.2
Rule SS10.	Selection Similarity	79	3.3.2.2
Rule SS10.	Component Selection Resolution	80	3.3.2.2
Rule IV1.	Invocation Resolution 1	85	3.3.3
Rule IV2.	Invocation Resolution 2	85	3.3.3
Rule SS11.	Invocation Similarity 1	88	3.3.3
Rule SS12.	Invocation Similarity 2	88	3.3.3

Bibliography

1. American National Standard COBOL, ANS X3.23-1974, American National Standards Institute, New York (1974).

2. American National Standard Programming Language Fortran, ANS X3.9-1978, American National Standards Institute, New York (1978).

3. Andrews, G.R. and J.R. McGraw, "Language Features for Process Interaction," Proceedings of ACM Conference on Language Design for Reliable Software, pp. 114-127 (March 1977).

4. Andrews, Gregory R., "Synchronizing Resources," ACM Transactions on Programming Languages and Systems Vol. 3(4) pp. 405-430 (October 1981).

5. Bloom, Toby, "Evaluating Synchronization Mechanisms," Proceedings of the 7th ACM Symposium on Operating System Principles, pp. 24-32 (December 1979).

6. Brinch Hansen, Per, "Structured Multiprogramming," Communications of the ACM Vol. 15(7) pp. 574-578 (July 1972).

7. Brinch Hansen, Per, Operating System Principles, Prentice Hall, Englewood Cliffs, N.J. (1973).

8. Brinch Hansen, Per, "The Programming Language Concurrent Pascal," IEEE Transactions on Software Engineering Vol. 1(2) pp. 199-206 (June 1975).

9. Brinch Hansen, Per, "Distributed Processes: A Concurrent Programming Concept," Communications of the ACM Vol. 21(11) pp. 934-941 (November 1978).

10. Campbell, R.H. and N. Habermann, "The Specification of Process Synchronization by Path Expressions," in Lecture Notes in Computer Science 16, Springer-Verlag (1974).

11. Chand, D.S., "On Applications of Data Abstraction Facilities," Proceedings of the ACM National Conference, pp. 639-645 (1978).

12. Chang, Ernest, Neil E. Kaden, and David W. Eliot, "Abstract Data Types in Euclid," SIGPLAN Notices Vol. 13(3) pp. 34-42 (March 1978).

13. Conradi, Reidar, "Some Comments on "Concurrent Readers and Writers"," Acta Informatica Vol. 8(4) pp. 335-340 (1977).

14. Conway, M.E., "Design of a Separable Transition-diagram Compiler," Communications of the ACM Vol. 6(7) pp. 396-408 (July 1963).

15. Courtois, P.J., F. Heymans, and D.L. Parnas, "Concurrent Control with "Readers" and "Writers"," Communications of the ACM Vol. 14(10) pp. 667-668 (October 1971).

16. Dahl, O-J., B. Myhraug, and K. Nygaard, "SIMULA 67 Common Base Language," S-22, Norwegian Computer Center, Oslo (1970).

17. Demers, Alan and James Donahue, "Revised Report on Russell," TR 79-389, Cornell University, Ithaca, New York (1979).

18. Dennis, Jack B. and Earl C. Van Horn , "Programming Semantics for Multiprogrammed Computations," Communications of the ACM Vol. 9(3) pp. 143-155 (March 1966).

19. Dijkstra, E.W., "Cooperating Sequential Processes," in Programming Languages, ed. F. Genuys,Academic Press, New York (1968).

20. Dijkstra, E.W., "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," Communications of the ACM Vol. 18(8) pp. 453-457 (August 1975).

21. Feldman, Jerome A., "High Level Programming for Distributed Computing," Communications of the ACM Vol. 22(6) pp. 353-368 (June 1979).

22. Geschke, Charles M., James H. Morris, and Edwin H. Satterthwaite, "Early Experience with Mesa," Communications of the ACM Vol. 20(8) pp. 540-553 (August 1977).

23. Gries, D.R. and N. Gehani, "Some Ideas on Data Types in High Level Languages," Communications of the ACM Vol. 20(6) pp. 414-420 (June 1977).

24. Griswold, R.E. and D.R. Hanson, "An Overview of SL5," SIGPLAN Notices Vol. 12(5) pp. 40-50 (April 1977).

25. Guttag, J., "Abstract Data Types and the Development of Data Structures," Communications of the ACM Vol. 20(6) pp. 396-404 (June 1977).

26. Guttag, J., E. Horowitz, and D. Musser, "Abstract Data Types and Software Validation," Communications of the ACM Vol. 21(12) pp. 1048-1064 (December 1978).

27. Hall, A.D., "The ALTRAN System for Rational Function Manipulation -- A Survey," Communications of the ACM Vol. 14(8) pp. 517-521 (August 1971).

28. Hanson, David R. and Ralph E. Griswold, "The SL5 Procedure Mechanism," Communications of the ACM Vol. 21(5) pp. 392-400 (May 1978).

29. Hecht, M.S., Data Flow Analysis of Computer Programs, American Elsevier, New York (1977).

30. Hoare, C.A.R., "Notes on Data Structuring," in Structure Programming, Dahl, O-J., Dijkstra, E.W. and Hoare, C.A.R.,, Academic Press (1972).

31. Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," Communications of the ACM Vol. 17(10) pp. 549-557 (Oct. 1974).

32. Hoare, C.A.R., "Communicating Sequential Processes," Communications of the ACM Vol. 21(8) pp. 666-677 (August 1978).

33. IBM System/360 Operating System: PL/I Language Specification (GC28-6571)., IBM Corporation, Data Processing Division, White Plains, New York (1965).

34. IBM System/370 Principles of Operations (GA22-7000-3)., IBM Corporation, Systems Products Division, Poughkeepsie, New York (1970).

35. Ingalls, D.H.H., "The Smalltalk-76 Programming System Design and Implementation," Conf. Rec. 5th ACM Symp. on Programming Language Principles, pp. 9-16 (1978).

36. Jones, Neil D. and Steven S. Muchnick, "Flow Analysis and Optimization of Lisp-like Structures," in Program Flow Analysis: Theory and Applications, Prentice Hall, Englewood Cliffs, New Jersey (1981).

37. Kaplan, M.A. and J.D. Ullman, "A General Scheme for the Automatic Inference of Variable Types," Conf. Rec. 5th ACM Symp. on Principles of Programming Languages, pp. 60-75 (January 1978).

38. Kennedy, K. and J. Schwartz, "An Introduction to the Set-theoretical Language SETL," in Computers and Mathematics with Applications, Pergamon Press (1975).

39. Kessels, J.L.W., "An Alternative to Event Queues for Synchronization in Monitors," Communications of the ACM Vol. 20(7) pp. 500-503 (July 1977).

40. Lampson, B.W., "Dynamic Protection Structures," pp. 27-38 in Proc. AFIPS 1969 FJCC, AFIPS Press, Montvale, New Jersey (1969).

41. Lampson, B.W., J.J. Horning, R.L. London, J.G. Mitchell, and G.J. Popek, "Report on the Programming Language Euclid," SIGPLAN Notices Vol. 12(2)(February 1977).

42. Lampson, B.W. and David D. Redell, "Experience with Processes and Monitors in Mesa," Communications of the ACM Vol. 23(2) pp. 105-117 (February 1980).

43. Lindstrom, Gary and Mary Lou Soffa, "Referencing and Retention in Block-structured Coroutines," ACM Transactions on Programming Languages and Systems Vol. 3(3) pp. 263-292 (July 1981).

44. Liskov, B. and S. Zilles, "Programming with Abstract Data Types," Proceedings of ACM Symposium on Very High Level Languages, SIGPLAN Vol. 9(4) pp. 50-59 (April 1974).

45. Liskov, B., A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," Communications of the ACM Vol. 20(8) pp. 564-576 (August 1977).

46. Liskov, B., E. Moss, C. Schaffert, R. Scheifler, and A. Snyder, "CLU Reference Manual," Computation Structures Group Memo 161, Laboratory for Computer Science, MIT (July, 1978).

47. Liskov, B., "Primitives for Distributed Computing," Proceedings of the 7th ACM Symposium on Operating System Principles, pp. 24-32 (December 1979).

48. Liskov, B., "On Linguistic Support for Distributed Programs," IEEE Transactions On Software Engineering Vol. 8(3) pp. 203-210 (May 1982).

49. Lohr, Klaus-Peter, "Beyond Concurrent Pascal," Proceedings of Sixth ACM Symp. on Operating System Principles, pp. 173-180 (November 1977).

50. Marlin, Christopher D., Coroutines: A Programming Methodology, A Language Design and an Implementation, Lecture Notes in Computer Science, Vol 95, Springer Verlag, New York (1980).

51. McGraw, James R. and Gregory R. Andrews, "Access Control in Parallel Programs," IEEE Transactions on Software Engineering Vol. 5(1) pp. 1-9 (January 1979).

52. McIlroy, M.D., Coroutines: Semantics in Search of a Syntax, unpublished manuscript 1968.

53. Morris, J., "Types are Not Sets," Proceedings of the ACM Symposium on the Principles of Programming Languages, pp. 120-124 (1973).

54. Morris, J.H., "Protection in Programming Languages," Communications of the ACM Vol. 16(1) pp. 15-21 (January 1973).

55. Naur, P. (ed.), "Revised Report on the Algorithmic Language ALGOL 60," Communications of the ACM Vol. 6(1) pp. 1-20 (January 1963).

56. Parnas, D.L., John E. Shore, and David Weiss, "Abstract Types Defined as Classes of Variables," Proceedings of Conference on Data: Abstraction, Definition and Structure, SIGPLAN Vol. 8(2) pp. 149-154 (March 1976).

57. "Preliminary Ada Reference Manual," SIGPLAN Notices Vol. 14(6)(1979).

58. Shaw, Mary, William A. Wulf, and Ralph L. London, "Abstraction and Verification in Alphard: Defining and Specifying Iterators and Generators," Communications of the ACM Vol. 20(8) pp. 553-564 (Aug. 1977).

59. Suzuki, Norihisa, "Inferring Types in Smalltalk," Conf. Rec. of the 8th ACM Symp. on Principles of Programming Languages, pp. 187-199 (January 1981).

60. Tennant, R.D., "A New Approach to Representations Independent Data Classes," Acta Informatica Vol. 8 pp. 315-324 (1977).

61. Tennant, R.D., "Another Look at Type Compatability in Pascal," Software: Practice and Experience Vol. 8(4) pp. 429-437 (July 1978).

62. Teorey, T.J. and T.B. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies," Communications of the ACM Vol. 15(3) pp. 177-184 (March 1972).

63. van den Bos, Jan, Rinus Plasmeijer, and Jan Stroet, "Process Communication Based on Input Specifications," ACM Transactions on Programming Languages and Systems

Vol. 3(3) pp. 224-250 (July 1981).

64. Wegbreit, B., "The Treatment of Data Types in EL1," Communications of the ACM Vol. 17(5) pp. 251-264 (May 1974).

65. Welsch, J., W.J. Sneeringer, and C.A.R. Hoare, "Ambiguities and Insecurities in Pascal," Software: Practice and Experience Vol. 7(6) pp. 685-696 (November 1977).

66. Wirth, N., "The Programming Language Pascal," Acta Informatica Vol. 1(1) pp. 35-63 (1971).

67. Wirth, N., "Modula: a Language for Modular Multiprogramming," Software Practice and Experience Vol. 7(1) pp. 3-35 (1977).

68. Wulf, W., R.K. Johnsson, C.B. Weinstock, S.O. Hobbs, and C.M. Geschke, The Design of an Optimizing Compiler, American Elsevier, New York (1975).

69. Wulf, William A., Paul Hilfinger, Gary Feldman, Robert Fitzgerald, Izumi Kimura, Ralph L. London, K.V.S. Prasad, V.R. Prasad, Jonathan Rosenberg, and Mary Shaw, "An Informal Definition of Alphard," CMU-CS-78-105, Carnegie-Mellon University, Pittsburgh, Pa. (February 1978).