Implementation and Evaluation of a Methodology for the Stepwise Refinement of Data

Boumediene Belkhouche

Computer Science Department
Tulane University
New Orleans, LA 70118

Implementation and Evaluation of a Methodology for the Stepwise Refinement of Data

Boumediene Belkhouche

Computer Science Department
Tulane University
New Orleans, LA 70118

A chunk structure was proposed as a program design technique to support the stepwise refinement of data structures in parallel with program development [12]. Both the definition of chunk structure and its required support induce new naming structures. This paper reports on an initial implementation of chunk structure and describes the basic mechanisms necessary to manage the name space of a program written in a language supporting such a feature. The evaluation of the technique is discussed in the context of design languages.

1. INTRODUCTION

The decomposition of complex programs [11,13] into manageable modules has improved the quality of software by providing well-structured and understandable systems. This methodology has essentially dealt with functional modularity; that is, modules representing operations are the units of the decomposition. Language issues such as side-effects, global variables [15], indiscriminate access to data and data structure complexity, among others, have motivated work related to data structures and have shown that data structuring is as important as program decomposition [5]. Moreover, the choice of data structures affects the complexity and efficiency of programs in which they are used [4]. Data abstraction as introduced in CLU [9], ALPHARD [16], MODEL [7], MODULA-2 [14] and Ada* [1] has provided the necessary data encapsulation; however, these languages do not provide the programmer with the ability to incrementally define data structures in parallel with the program design. The duality of programs and data structures dictates that the concept of stepwise refinement of programs be extended to data structures. The capability to incrementally refine data structures would prevent precocious bindings and allow decisions about data structures to be deferred until such time in the development that they need be made.

A data structure methodology which provides the designer with the capability of making these decisions at a more appropriate stage is described section 2. An implementation model that captures the naming structures induced by chunk structure is illustrated in section 3. This model was abstracted from an interpreter implemented by the author. The evaluation of the methodology is exposed in section 4.

2. FEATURES OF CHUNK STRUCTURE

1. The use of incomplete declarations: a data name may be declared without being bound to structure or type attributes.

As can be seen, some components of "employee_record" are not required in this module and are omitted in the declaration. This example illustrates the following three concepts:

1. The use of incomplete declarations: a data name may be declared without being bound to structure or type attributes.

In this instance, the data name has been refined to include four components. Yet, in another module that computes the salary, the following declaration may be necessary:

(3) var employee_record: record
    hours: integer;
    rate: real;
    salary: real
end;

As can be seen, some components of "employee_record" are not required in this module and are omitted in the declaration. This example illustrates the following three concepts:

1. The use of incomplete declarations: a data name may be declared without being bound to structure or type attributes.

...
The refinement of an incomplete declaration at a later stage: structure or type attributes may be associated with a data name in subsequent modules, i.e., information about the data name is refined.

3. Different modules may have distinct logical views of a data name: modules that refine the same data name differently have a different view of that name.

A data structuring technique, called chunk structure, was proposed by Towster [12] to support these concepts. The technique is independent of other features of the base programming language supporting chunk structure. To allow for the top-down refinement of data structures, abstract data names (also called chunk names) are introduced. In the previous example, the data name "employee-record" is an instance of a chunk name. Data names that are conceptually related are defined as refinements of an abstract data name. At an early stage of program development, we may choose to define only abstract data names. As the top-down design of modules proceeds, these abstract data names are similarly further refined by associating with them a group of new data names. The refinement process for a given data name ends whenever a type attribute is bound to that name, which then becomes a concrete data name (e.g., "name" and "ssn" are concrete data names; they can not be refined any further). Pictorially, an abstract data name can be viewed as the root of a subtree. Its descendants constitute a refinement and may themselves be abstract data names representing some conceptual subgroups subject to further refinements. The terminal nodes of a completely refined subtree represent concrete data names. The subtree representing an abstract data name need not be fully defined at any point during program development. Yet, the program can still be executed.

A program expressed in a chunk structured language consists of a set of non-nested module definitions each of which represents a functional abstraction. A module has three parts: a title, an environment description section (EDS), and an executable code section (ECS). The title serves to identify the module and to specify the name of the parameters and their order. The EDS defines the set of all data names which may be referenced within the module. For each name, it specifies which details of the structure of that name are required by the module in order to perform its function. All parameters must appear within this section; all locally defined names are also included. The ECS is expressed as a sequence of operations, and is analogous to a block. It consists of a sequence of statements supported by the base language. Referencing within the ECS is limited to only those names defined in the corresponding EDS.

To illustrate some of the features of chunk structure more concretely, an example of a program skeleton written in a language designed to support chunk structures is discussed. The program is shown in Figure 1. (See references [12] and [3] for a more detailed treatment of the language.) For ease of description, line numbers are displayed. A program is a set of non-nested modules, each delimited by the keywords begin and end. The title is the text enclosed between quotes following the keyword begin. Parameters of the module appear within its title and are prefixed and/or suffixed with a the symbol "<-" (rules for placement are given below). The EDS is delimited by the keywords env and end env. The refinements of abstract data names appearing in the EDS are indicated by indentation [8].
In the first EDS (lines 2-6), the data name "master-data" is refined to consist of an integer, "number-of-employees", and an array, "employee". The details of an element of the data name "employee" have not been defined since no refinement is associated with it. The slash (/) prefixing "master-data" is called a local mark and serves to distinguish locally-defined data names from passed data names (formal parameters).

The second EDS (lines 13-21) defines the formal parameters "x" as an integer, and "employee" as an array. Each element of the array is newly refined to consist of two string data names, "ssn" and "name", and two integer data names, "rate" and "hours". The data name, "count", which is part of this EDS, is a local integer data name. Thus, this module has two distinct referencing environments: a local environment and a passed environment.

The local environment defined in the third EDS (lines 24-32) is composed of the integer data name, "in", and the passed environment consists of the data name "master-data". This data name is refined to include an integer data name, "number-of-employees", and the array data name, "employee". This array has three data names as part of its refinement: the data name "salary", which is a new refinement, and the two other data names, "hours" and "rate", which appear already in the second EDS. These refinements are not positional but rather by name. The ability of chunk structure to allow different modules to exhibit identical refinements is used as a means of communication among these modules. This mechanism permits lower modules to share information without requiring that information be known more globally as is the case in block-structured languages.

In each of the modules in this example, the data name "employee" is refined differently to include only the information needed by that specific module. Each module has a different logical view of the data name "employee", and a given module has access to only those data names it defines in its EDS; the other data names are not visible to it.

Access rights are associated with each data name declared in the EDS by means of a slash (/) prefixing a data name, or a chunk mark (<-) prefixing and/or suffixing a data name. The semantics of access rights association is defined as follows:

1. /a : "a" is a local data name. The module in which "a" is defined may retrieve the current value of "a", and may alter that value.
2. <-a : "a" is a constant data name. The module to which "a" is passed as a parameter may only retrieve the value of "a".
3. a<- : "a" is a virgin data name. The module to which "a" is passed as a parameter has the responsibility to initialize "a".
4. <a<- : "a" is a modifiable data name. The module to which "a" is passed as a parameter may retrieve and alter the value of "a".

Cases (2), (3) and (4) are basically similar to IN, OUT and INOUT parameters of Ada [1].

3. NAMING STRUCTURES

Towster's proposal dealt with the methodology and did not address implementation issues. A language supporting chunk structure was designed, and an interpreter and a compiler for this language was implemented on a MULTICS system to investigate these issues. The mechanisms of the interpreter that were developed to manage dynamically the name space of a program are described in the following sections.

The requirement that all environments be declared explicitly and the capability to refine abstract names incrementally induce new naming structures. Each module owns and has access to only one symbol table but, because of mutual refinements of identical abstract names, portions of the symbol tables may be shared by different modules. This view is different from other approaches where either symbol tables are included within each other (block-structured languages), or they are completely separated (flat languages). Symbol tables in chunk structure overlap and it is through their intersection that modules communicate. Two basic mechanisms must be provided to manage the program name space. They are:

1. The maintenance of the relationship among interacting modules and among abstract names and their refinements. A global symbol table, called the Chunk Tree, is used to maintain this relationship. The Chunk Tree grows and shrinks dynamically.

2. The definition of the name space that a module can manipulate. A module symbol table, called the Reference Region, defines the referencing environment of a module.

3.1 THE CHUNK TREE

The chunk tree defines the total active name space of a program during execution. The
relationship among interacting modules is tree-like; i.e., the calling module is considered the parent of the called module. There is also a relationship between a module and the local data names it defines. This relationship is reflected in the chunk tree by the module node being the parent of the subtrees of the data names that are declared locally. Conceptually, every module has a corresponding node that is the root of a subtree. Data names defined in its EDS are the immediate descendants of the root. Nodes for other modules that are invoked from this module are also direct children of the root node. Therefore, building the chunk tree proceeds as follows. The tree root node corresponding to the main module is created. Subtrees representing data names declared in the first EDS are added to the tree as direct descendants of the tree root. If, and when, a module is invoked, a corresponding node is added to the tree as a child of the calling module. Then, the EDS of the called module is processed. When a module is exited, its subtree is deleted. That is, the tree is dynamic in nature so that, at any moment of execution, it reflects only the total active name space. The chunk tree is specifically used to enforce the consistency of the various refinements of an abstract data name, and to identify modules in which the first refinement of a name appears. A refinement to a local data name involves updating the information in the chunk tree node being refined and/or adding descendant nodes. Refinements to names that are passed from one module to another require checking for compatibility and consistency among the different refinements. No action is performed if the refinement already exists in the tree (i.e., a previous module made the same refinement). Otherwise, the tree is updated to reflect the new refinement.

3.2 THE REFERENCE REGION

The EDS defines statically the referencing environment of a module [10]. This environment consists of all locally defined names and parameters. Each data name appearing in the EDS is represented as a subtree, the collection of which forms a local data structure called the reference region. This is used to define a one-to-one mapping between names in the EDS and nodes in the chunk tree, the mapping determines the accessible name space in the chunk tree. A reference region is associated with each active module; however, a module has access to its reference region only. This policy is enforced through a stack-based scheme. Consequently, a data name reference in an ECS is resolved successfully only if it has a corresponding node in the currently active reference region.

The lifetime of a data name is defined in terms of its presence in the chunk tree. A node for a data name is created in the chunk tree when this data name is first encountered in a reference region, and is destroyed only when the module under which it is attached is deactivated. The effect of module deactivation is to destroy the reference region for that module and to destroy all the nodes in the subtree of the chunk tree whose root is the node for that module.

The scope of a name is defined in terms of its accessibility. Since a module can access a name only if that name is in the reference region of the module, the scope of a name is defined by the set of Reference Regions in which it appears.

3.3 Illustration

In the following discussion, examples to illustrate the relationship between the EDS, the Reference Region and the Chunk Tree are presented. These examples are solely used to provide a basis for constructing and pruning the chunk tree. Several figures representing the chunk tree at key points of the program are included. These key points are identified as the end of an EDS, and the exit of a module.

```
1 begin "Example to illustrate local environment";
2 env;
3 /pastime
4 /date string
5 /employee string
6 name string
7 education
8 end env;
9 "Refine pastime<-, and initialize date<-
   and name in employee<-";
10 end;
```

Figure 2: Example 1

Scanning the EDS, we notice five different types of data name definitions. Thus, on line 3, the data name *pastime* is an abstract name, since it does not have any type information associated with it. On line 4, *data* is a concrete data name, since it can assume values of type "string". On line 5, *employee* is an abstract data name which has been refined. As part of its refinement, we have two data names: *name* which is a concrete data name, and *education* which is still abstract. Both data names appear indented in the EDS. Indentation is used to convey the relationship that is reflected in the Chunk Tree by the relation parent-child, with the indented data names being the children of the abstract
name. The parent of the unindented data names is implicit and is the name of the module itself, which means that "pastime", "date", "employee" have as parent the module node. Consequently, the Chunk Tree will have a node for the module; this node will have three nodes as children, and the node standing for "employee" will have two nodes as children. Figure 3 illustrates the portion of Chunk Tree for this example. In all the figures, only the relationship between names is displayed. The node "example 1" represents the module.

Figure 3: Chunk Tree for the module of Figure 2 at line 8

The Reference Region is the set of subtrees which represent the active names of the module. This implies that the Reference Region is a forest, each tree of which represents an abstract name with its refinements. Here, too, the parent-child relation captures the relationship between data names as defined in the EDS. The role of the Reference Region is to designate that part of the Chunk Tree which is accessible to a module. Each node in the Reference Region is mapped directly into one node in the Chunk Tree. Figure 4 illustrates the Reference Region for Example 1.

Figure 4: Reference Region for the module of Figure 2

A module may also have a passed environment. Initially, this environment is specified by the set of abstract names which appear in its title. Within the EDS, these same names are restated, along with their accompanying chunk marks. Further refinements to these names may be performed to augment the passed environment of the module with data names that were not explicitly passed.

The EDS in Figure 5 differs from that of Figure 2 in that it has two environments: a passed environment and its refinements, and a local environment. The Reference Region will still be a mapping of the EDS, as shown in Figure 6, but the effects of this EDS on the Chunk Tree are different. These effects will be described later.

1 begin 'Refine Hobbies<- and initialize
2 Date<- and name in Employee<-';
3 env;
4 Date<- string
5 Hobbies<- string
7 /error boolean
8 Employee<- string
9 name
10 and env;
11 "Restructure <-name and <-music[1].";
12 end;

Figure 5: Example 2

The second component of a Data Chunking module is the executable code section or ECS. It consists of a sequence of statements of any type supported by the base language. Within the ECS, only names defined in the corresponding EDS can be referenced. One statement type peculiar to a Data Chunking language is the module reference statement. Line 9 in Figure 2 is such a statement, and generates the interaction between the first and the second module as illustrated in the example in Figure 7.
begin "Example to illustrate local environment";
env;
pastime
date string
employee name string
educationend env;

"Refine pastime<-, and initialize date< and name in employee<";
end;

begin "Refine Hobbies<-, and initialize Date< and name in Employee<";
env;
Date<- string
Hobbies<- string
music [2] string
/error boolean
Employee<- name string
end env;

"Restructure <-name and <-music[l].";
end;

Figure 7: Example 3

Module invocation is caused by a module reference statement, similar to call statements in other programming languages. A module reference is an instance of the title of the submodule to be invoked. When the second module is activated as a result of the execution of line 9 in Figure 7, information from the first module is passed to it. This information represents the passed environment, and each of its elements is recognized in the title and in the EDS by the chunk mark that is concatenated to the data name. On line 9, the passed environment is comprised of "pastime", "date", and "employee", and constitutes the list of formal parameters. The passed environment, which may consist of expression values as well as data names, forms the subset of the Reference Region of the called module. Within the EDS (lines 12-20, Figure 7), the passed environment and its refinements are listed. The abstract name "Hobbies" is refined to include two concrete data names, one of which is an array. The effect of this refinement on the Chunk Tree is to add a scalar terminal node for "ski" and an array terminal node for "music" under the node for the parameter "Hobbies", which is the same as the node that was added earlier in the Chunk Tree to represent the abstract name "pastime". At this point, any module having access to "pastime" may also access "ski" and "music" if they are listed in its EDS. Although, "Employee" represents "name" and "education" in the Chunk Tree, an abstract reference to "Employee" in this module will access "name" only, because the Reference Region is used to resolve references. Therefore, "education" is invisible to this module. Additions of nodes in the Chunk Tree to represent refinements are made only the first time a refinement is defined. For this reason, the passed environment is concise because of the ability of Data Chunking to pass abstract names. Refinements to formal parameters have no effect on the Reference Region of the calling module. The abstract name "error", which is defined in line 17, is a local name for this module. Figure 8 illustrates the Chunk Tree for the example of Figure 7 as it exists during execution of the second module. Figure 9 represents the Reference Region for the second module in Figure 7. The reader should note the implicit definition of "music [1]" and "music [2]" displayed in Figure 8.

Figure 7: Example 3

Module invocation is caused by a module reference statement, similar to call statements in other programming languages. A module reference is an instance of the title of the submodule to be invoked. When the second module is activated as a result of the execution of line 9 in Figure 7, information from the first module is passed to it. This information represents the passed environment, and each of its elements is recognized in the title and in the EDS by the chunk mark that is concatenated to the data name. On line 9, the passed environment is comprised of "pastime", "date", and "employee", and constitutes the list of formal parameters. The passed environment, which may consist of expression values as well as data names, forms the subset of the Reference Region of the called module. Within the EDS (lines 12-20, Figure 7), the passed environment and its refinements are listed. The abstract name "Hobbies" is refined to include two concrete data names, one of which is an array. The effect of this refinement on the Chunk Tree is to add a scalar terminal node for "ski" and an array terminal node for "music" under the node for the parameter "Hobbies", which is the same as the node that was added earlier in the Chunk Tree to represent the abstract name "pastime". At this point, any module having access to "pastime" may also access "ski" and "music" if they are listed in its EDS. Although, "Employee" represents "name" and "education" in the Chunk Tree, an abstract reference to "Employee" in this module will access "name" only, because the Reference Region is used to resolve references. Therefore, "education" is invisible to this module. Additions of nodes in the Chunk Tree to represent refinements are made only the first time a refinement is defined. For this reason, the passed environment is concise because of the ability of Data Chunking to pass abstract names. Refinements to formal parameters have no effect on the Reference Region of the calling module. The abstract name "error", which is defined in line 17, is a local name for this module. Figure 8 illustrates the Chunk Tree for the example of Figure 7 as it exists during execution of the second module. Figure 9 represents the Reference Region for the second module in Figure 7. The reader should note the implicit definition of "music [1]" and "music [2]" displayed in Figure 8.

Figure 7: Example 3

Module invocation is caused by a module reference statement, similar to call statements in other programming languages. A module reference is an instance of the title of the submodule to be invoked. When the second module is activated as a result of the execution of line 9 in Figure 7, information from the first module is passed to it. This information represents the passed environment, and each of its elements is recognized in the title and in the EDS by the chunk mark that is concatenated to the data name. On line 9, the passed environment is comprised of "pastime", "date", and "employee", and constitutes the list of formal parameters. The passed environment, which may consist of expression values as well as data names, forms the subset of the Reference Region of the called module. Within the EDS (lines 12-20, Figure 7), the passed environment and its refinements are listed. The abstract name "Hobbies" is refined to include two concrete data names, one of which is an array. The effect of this refinement on the Chunk Tree is to add a scalar terminal node for "ski" and an array terminal node for "music" under the node for the parameter "Hobbies", which is the same as the node that was added earlier in the Chunk Tree to represent the abstract name "pastime". At this point, any module having access to "pastime" may also access "ski" and "music" if they are listed in its EDS. Although, "Employee" represents "name" and "education" in the Chunk Tree, an abstract reference to "Employee" in this module will access "name" only, because the Reference Region is used to resolve references. Therefore, "education" is invisible to this module. Additions of nodes in the Chunk Tree to represent refinements are made only the first time a refinement is defined. For this reason, the passed environment is concise because of the ability of Data Chunking to pass abstract names. Refinements to formal parameters have no effect on the Reference Region of the calling module. The abstract name "error", which is defined in line 17, is a local name for this module. Figure 8 illustrates the Chunk Tree for the example of Figure 7 as it exists during execution of the second module. Figure 9 represents the Reference Region for the second module in Figure 7. The reader should note the implicit definition of "music [1]" and "music [2]" displayed in Figure 8.
in the Chunk Tree and the Reference Region. However, a submodule need not be explicitly listed in the EDS. In case it is, it is typed as an entry, and is a constant concrete data name. If it is not listed in the EDS, the submodule is considered to be an implicit direct descendant of the calling module. The effect of executing line 9 is to add a module node for this called module as a child of the calling module, and to add a node in the Reference Region as a sibling of the unindented data names in the EDS. Then, the called module is activated.

In comparing the module reference title on line 9 with the heading title on line 11, we note that they are identical except for the names in the passed environment. These names may or may not be the same as long as the position of the chunk mark is maintained. Moreover, positional correspondence rules are used to bind the actual and the formal parameters.

Execution of line 22 results in the exit of the sub-module. Its Reference Region as well as its local environment are destroyed. However, before exit from this module, another module reference is encountered in line 21. Expanding Example 3 by defining this third module yields Example 4 (Figure 10), which illustrates some additional features of the Data Chunking naming structure.

In the EDS described in lines 24-28 of Figure 10, a relationship between a person and a type of music is implied. Although these two data names come from different parts of the Chunk Tree, they are conceptually related in this module, and could be grouped as a part of a local abstract name. This process is called restructuring, a feature that enables the Chunk Tree to be reconfigured to suit some new conceptual view. The effect on the Chunk Tree is to create a local node for "music-lover", and to move the nodes for "name" and "music[1]" (the actual parameters) making them children of "music lover". This configuration remains in effect until this module is exited, at which time the Chunk Tree will be returned to the form it had before restructuring.

In line 21 of Figure 10, a portion of an array was passed to a submodule. This illustrates the fact that a form of cross-sectioning is supported by Data Chunking. Unlike other naming structures, such a cross-section may be refined within the submodule, thereby defining the structure of the array elements. Such is the case when an element of an abstract array is passed to a submodule. When a cross-section of a completely refined array is passed to a submodule, the effect is well defined and analogous to using an array element as an actual parameter in more traditional naming structures. The current implementation requires that all array elements be homogeneous, even though they are not visible. Therefore, refinement of a cross section implies refinement of all the elements of the array.

begin "Example to illustrate local environment";
  env;
  /pastime string
  /date string
  /employee string
  name string
  education
  end env;

  "Refine pastime<-, and initialize date<-
  and name in employee<-";
  end

begin "Refine Hobbies<-, and initialize
  Data<-, and name in Employee<-";
  env;
  Data<- string
  Hobbies<- string
  /music[2] string
  /error boolean
  Employee<- string
  name string
  end env;

  "Restructure <-name and
  <-music[1].";
  end

begin "Restructure <-person_name and
  <-music_type.";
  env;
  /music_lover
  <-person_name string
  <-music_type string
  end env;

Figure 10: Example 4

The concept of abstract names is a powerful one. Data Chunking associates with it the concept of abstract operations, which use abstract names as operands. This implies a simplification in the program, and allows for concise reference within the ECS. Operations for which this operation extension is meaningful, such as input/output, assignment, and comparison for equality, can be applied to abstract names. For an abstract binary operation, it is required that the operands be isomorphic up to renaming; i.e., the corresponding names must be of the same type, and the trees associated with the abstract names must be of the same shape. Since these names stand for all the accessible concrete data names under them, the result is to apply the operation to each concrete data name.
The reference region and the chunk tree for the example of Figure 10 are shown in Figures 11 and 12, respectively.

4. EVALUATION OF CHUNK STRUCTURE

The evaluation of chunk structure is discussed in the context of design languages, and it is assumed that the criteria of efficiency of compilation and execution are not critical in this context. This does not exclude executable design languages that allow the designer to test the design before defining an implementation.

A design language supporting chunk structure will have several interesting features. This language will allow the designer to devise data structures without binding them to any specific implementation. The focus during the design will be on progressively discovering the relationship between the data. As this relationship becomes more apparent, the operations on the data are defined accordingly. This incremental definition seems to be a methodology for defining abstract data types (given that an abstract data type is defined only after the relationship between the data and the operations is defined).

Embedding chunk structure in an executable design language does create some problems. Of particular concern is the scope of a module; i.e., the accessibility of a module title. Intuitively, since the module is the mechanism by which procedural refinement is supported, access to the module title should be localized. However, since module definitions are not nested in the program, no explicit method of delimiting the scope of the module title is defined. Once the node representing the module has been placed in the Chunk Tree, its immediate ancestor has been determined and its scope can be limited to that ancestor alone. This formulation eliminates recursion. Chunk structure can be modified to allow recursion by asserting that a module has access to the titles of all modules in the path from the module node to the root of the Chunk Tree, but this view seems antithetical to the requirement of explicit environment description for access to names. As a result, the definition of chunk structure must be modified to explicitly address this issue.

A second difficulty concerning the scope of the module is that module definitions cannot be shared. This encourages "reinventing the wheel", which is undesirable. To eliminate this problem, the notion of entry data was proposed by Towster [12]. A module title can be made accessible to several modules by passing it as a parameter, in the same way that abstract names can be shared by several modules. The parameter corresponding to a module argument is of type entry. There are several problems with this proposal. The first is that it becomes extremely difficult to algorithmically and unambiguously identify the definition of an entry type module within the program stream. The second is that access to module titles becomes inconsistent, sometimes determined by the EDS and sometimes determined solely by the ECS. To accommodate recursion, the module must be explicitly passed its own title as an argument. This negates the advantages of functional refinement because it implies prior knowledge of the details of that module, since the decision to employ a recursive algorithm must be reflected within the parameterization.

Finally, there is a problem concerning data aggregation. Semantically, one expects array elements to be homogeneous at the implementation.
This is not the case in chunk structure where homogeneity is not required. Abstract arrays are treated as arrays of variant records [6]. Each element of the array can be refined differently from the other elements. This flexibility might require a thorough analysis of the design to determine whether, at the implementation level, a conventional array addressing technique or a more complex addressing scheme is to be used to define the mapping between the abstract array and its concrete representation.

The difficulties with nonhomogeneous arrays are real only at the implementation language level. The design language need not have all of the constructs available in the implementation language. The above shortcomings can be easily overcome during the translation process from the design to the implementation. The translation would involve the selection of representations of data structures. These representations may be quite different from what has been defined in the design, as long as the meaning is maintained.

5. CONCLUSION

Chunk structure concepts resulted from considerations of problems faced by programmers using existing programming languages. Problems involving the design of data structures and the vulnerability of data led to the introduction of the concepts of stepwise refinement of data and explicit environment description. The implementation model described in this paper delineates the basic capabilities which a chunk structure processor must provide. This model was abstracted from the interpreter implemented on MULTICS to process TODDLER, a language which supports chunk structure [2, 3].

Given the complexity of the underlying support required for chunk structure, it is not expected that a chunk structured language could be directly used in a production environment. Rather, such a language would appear to be useful as a design tool (e.g., Ada PDL) embedded within a comprehensive software development system in which testing of incomplete programs can be performed.

ACKNOWLEDGEMENTS

I would like to thank Pat Carr and Ed Towster for their contributions to this work. I would also like to thank Greg Riccardi for his comments on earlier versions of this paper. The comments of two anonymous referees helped improve the exposition of the material. I would like to thank both of them. The implementation of the language processor was partially supported by NSF grant MCS78-02690.

REFERENCES