

Using IDL in a Heterogeneous Environment

TR88-040

August 1988

Sundar Varadarajan

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175



UNC is an Equal Opportunity/Affirmative Action Institution.

Using IDL in a Heterogeneous Environment

by

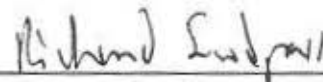
Sundar Varadarajan

A Thesis submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

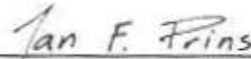
Chapel Hill

August 1988

Approved by:



Advisor - Dr. Richard Snodgrass



Reader - Dr. Jan Prins



Reader - Dr. Dean Brock

©1988
Sundar Varadarajan
ALL RIGHTS RESERVED

Acknowledgements

I would like to thank Rick Snodgrass for his help and encouragement through the course of this research. I would like to thank Jan Prins and Dean Brock for taking their time to serve on the thesis committee and for their comments on this text. I would like to thank Ed Mckenzie for his advice on various matters.

To
Appa and Manni

Contents

1	Introduction	1
1.1	Problem	3
2	Previous Work	5
3	The Methodology	7
3.1	Dimensions Of the Problem Space	10
3.2	Approaches	12
3.3	Metrics	18
3.4	Mapping	18
3.5	Metrics on the Solutions	20
3.6	Other Techniques	27
3.7	Transforming Many Interfaces	28
3.8	Conclusions	29
4	Transforming a Syntax-directed Editor	30
4.1	Transformation of the output interface	32
4.2	Transformation of the input interface	35
4.3	Conclusions	38
5	Transforming XDR	39
5.1	Transformation of the Input Interface	41
5.2	Transformation of the Output Interface	43
5.3	Conclusions	45
6	Conclusions	47
6.1	Future Work	47
7	Bibliography	49
A	Glossary	51
B	Code for Syntax-directed Editor	54
B.1	The Specification	54
B.2	Transformation of the Output Interface	57
B.3	Transformation of the Input Interface	62

C	Code for XDR	74
C.1	The Specification	74
C.2	Input and Output using XDR	77
C.3	Transformation of the Input Interface	80
C.4	Transformation of the Output Interface	84
D	Summary of the Methodology	87

Chapter 1

Introduction

One of the major problems in Computer Science concerns the development and maintenance of complex software systems. A method of development is to divide the system into smaller subsystems that interact with each other and the environment in a well-defined fashion. These subsystems could then cooperate to achieve the goals of the computer system. While such a division is neither trivial nor obvious, once it has been done the subsystems can be developed independently and concurrently. Parallel implementation assumes that the subsystem, as well as its interaction with other subsystems and the environment, is completely specified. Therefore, we need a framework to specify the subsystem.

A subsystem is specified when its interfaces and the transformation it performs on the input data to compute the output data is specified. This specification can be used to develop and test the subsystem. It is conceivable that a subsystem can be generated from such a specification. The specification for a subsystem consists of a part that specifies the structure of the data at the interfaces and another that specifies the transformations that it performs. The interface data structure specification can be used to generate data to test the subsystem. The subsystem reads and writes instances of these data structures. The mapping from these instances to a form that is communicable across the physical interface can be done automatically.

The Interface Description Language (IDL) [Nestor et al. 1982] is a language for specifying the structure of the data communicated across the interfaces. IDL is a data structure specification language and includes a set of basic types and constructors that may be used to construct new types from other types. The basic types are Integer, String, Boolean and Rational. A *node* in IDL is analogous to a Pascal [Wirth 1971] record. The *attributes* of a node are analogous to the fields of a record. Some attribute values may be shared between two or more nodes. Sets and Sequences of a type may also be constructed. A class is a group of nodes that may have some common attributes. Every structure specification has a distinguished node (or class) called the *root* from which all the other nodes in the structure may be accessed. For example, consider the specification in Figure 1.1, that specifies an expression tree. The root of the structure 'exp' is a class that has as its subclasses 'binop_exp', 'unop_exp' and 'term'. The class 'exp' has associated with it the attribute 'value' of type Integer. The class 'binop_exp' has two attributes 'left_exp' and 'right_exp', that


```

Structure exp_tree Root exp Is
  exp      ::= binop_exp | unop_exp | term ;
  exp      => value: Integer ;
  binop_exp ::= add_exp | sub_exp | mult_exp | div_exp ;
  binop_exp => left_exp: exp,
              right_exp: exp ;
  unop_exp ::= negate_exp ;
  unop_exp => expr: exp;
End

```

Figure 1.1: An IDL specification for an expression tree

```

add_exp
  [value 5;
   left_exp term
     [value 2];
   right_exp L1^
  ]
L1: term
  [value 3]
#

```

Figure 1.2: ASCII ERL representation of an expression tree

are common to all sub-classes of that class. 'term' is a node that has no attributes. IDL can be used to specify directed, and possibly cyclic, graphs.

The IDL Toolkit [Snodgrass 1988] provides a number of tools to facilitate the use of IDL on subsystems developed for use with the UNIX operating system. In this realization of IDL, each subsystem is mapped to a UNIX process. The subsystems interact with each other through data communicated through files (or through pipes). The data, which are an instance of the IDL specification, are communicated as a sequence of ASCII characters in a format called the ASCII External Representation Language or ASCII ERL.

The ASCII ERL representation of a structure is the ASCII ERL representation of the root of the structure. The ASCII ERL representation of a node is the name of the type of the node concatenated with enumeration of all the attributes of the node. The ASCII ERL representation of the expression tree for the expression '2+3' is given in Figure 1.2. This expression tree is an instance of the IDL specification in Figure 1.1. Each attribute of the node is listed as an (attribute name, attribute value) pair (e.g., value 3). Instead, the attribute may be listed as a (attribute name, attribute label) pair (e.g., right_exp L1^). This label refers to the value of the attribute that occurs

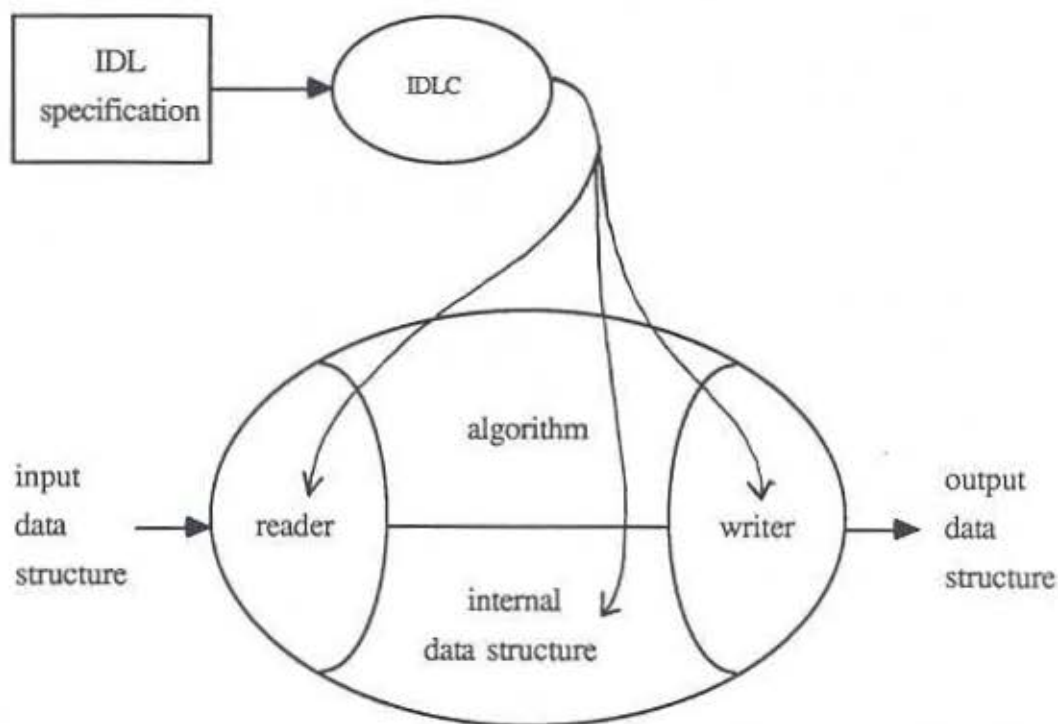


Figure 1.3: An IDL generated process

elsewhere in the IDL instance.

The idl translator (IDLC), one of tools in the toolkit, maps IDL specifications into data declarations in a target language, i.e., a programming language like 'C'. IDLC also generates readers, that create the IDL instance in memory from the data in ASCII ERL format. Similarly, writers are generated to write out the in-memory IDL instance in the ASCII ERL format. A tool implemented using the toolkit reads in the IDL instance using the readers, computes the output instance and writes out the IDL instance using the writers. This is illustrated in Figure 1.3. The IDL instance output can be read by another process. Communication of complex data structures is fairly simple using the readers and writers provided by IDLC.

1.1 Problem

A tool may be specified in IDL by specifying the structure of its input and output data in IDL. There are a number of existing tools (processes) which are not written using the toolkit, but whose input and output can be specified in IDL. The format in which the input and output data are stored externally are specific to each tool. Even

if two processes input and output the same data structure, they may not be able to communicate with each other, since the formats may be different. This research will consider ways to enable such processes to input/output the IDL instances to other processes using the toolkit.

The basic problem is one of using IDL in a heterogeneous environment, an environment in which some processes use IDLC and some do not. First one has to specify the process' input and output in IDL. Then the process should communicate instances of this specification in a format acceptable to other processes. The problem of communicating instances in a format acceptable to other processes is the main focus of this research.

There are two solutions to this problem. One is to modify IDLC to generate code that accepts the data produced by the process as a representation of an IDL instance. Another is to modify the process to input and output the IDL instance in the standard format. Consider a situation in which n processes output the same IDL instance. In the first case, n different processes would produce n representations for the same IDL instance. If m tools accepted this IDL instance, n versions of each tool would be needed, requiring $m * n$ tools. In the second case, n processes would have to be modified and only m tools would be needed. The idea is to convert the IDL instance to the standard format expected by IDLC so that any process written using IDLC can use the information.

This research considers the problem of transforming the process (or its input and output) so that data structure it communicates will be in the standard format. There are a few approaches to performing such a transformation. A methodology is proposed that may be used to choose between the various approaches depending on the characteristics of the problem. Metrics associated with the different approaches are provided. The methodology is tested by applying it to transform two tools.

Related work done in the area of higher level data interfaces is considered in the next chapter. Chapter 3 presents the different approaches, the methodology to choose between the different approaches and the metrics associated with the different approaches. The methodology is applied to two different tools, a syntax directed editor, discussed in Chapter 4 and XDR, discussed in Chapter 5. Chapter 6 presents the results of this research and points to future work in the area.

Chapter 2

Previous Work

The area of higher level data interfaces overlaps two areas: computer communication, concerned with the transfer of data, and interprocess communication, concerned with issues in exchanging messages between processes. This chapter examines the work done in these areas with emphasis on higher level data interfaces.

The advent of computer networks has given rise to a number of communication protocols. The International Standards Organization has developed a seven-layered model of a computer network called the Reference Model of Open Systems Interconnection [Tanenbaum 1981A]. Most of the existing communication protocols [Tanenbaum 1981B] are transport, network and data link layer protocols (e.g., HDLC, X.25). These layers are concerned with the issues of routing and the unreliability of the physical medium. The session, presentation and application layers are concerned with establishing communication, performing transformations on data (e.g., data compression and encryption) and supporting specific applications. Most existing high level protocols like virtual terminal protocols, file transfer protocols and message transfer protocols are geared to very specific applications. The ASCII ERL could form the basis of a protocol to communicate complex data structures between processes on a computer network.

Interprocess communication is concerned with issues in exchanging messages between processes. The emphasis in research has been the sharing of resources between processes. There has been a lot of work in process synchronization, atomicity and serialization of operations. There are a number of algorithms and models that resolve some of these issues [Chambers, et al. 1984].

There has not been much work in the area of higher level data interfaces. Most data interfaces (e.g., UNIX pipes) are organized as a stream of bytes or characters. Some data representation protocols are capable of communicating complex data structures. An example of this is the External Data Representation Scheme (XDR) [XDR 1986] developed by Sun Microsystems Inc. XDR is a procedural interface that can be used to communicate between processes', data structures specified in the 'C' programming language [Kernighan & Ritchie 1988]. However, data structures communicated using XDR are represented in-memory and therefore cannot be stored in files as data communicated using pipes can.

IDL is a language for specifying data structures communicated between processes.

The IDL Toolkit, a set of tools to facilitate the use of IDL, represents data structures externally in the ASCII ERL format. Biyani [Biyani 1987] has developed a run-time system for IDL that represents IDL data structures in a more compact, but language and machine dependent, format.

Chapter 3

The Methodology

Consider a tool that reads data, performs some computation and writes out data as shown in the Figure 3.1. The input and output data are in a format specific to the tool. The routines that read and write external data, and the implied format of that data, is termed the tool's *interface*. It is desired that this tool be converted so that its input and output data are instances of IDL structures as shown in Figure 3.2. That is, all the interfaces need to be *transformed* to read and write IDL instances.

This chapter discusses a methodology to transform an interface so that it may read or write instances in the standardized IDL External Representation Language, or simply the *IDL format*. In general, more than one interface may need to be transformed. The methodology that is proposed here will aid the transformation of one such interface at a time. However all the interfaces may be transformed by applying the methodology separately to each one of the interfaces. Thus a tool with several interfaces can be transformed to input and output IDL instances. Such a tool would input and output IDL instances in the standardized format.

The use of a standardized data format (the IDL format) gives rise to the possibility that several tools may be interfaced together. For example, a code-generator can be built that can generate code from the frontends (consisting of syntactic and semantic analysis phases) of both Pascal and C compilers. At first sight, it appears that the transformation of the interface between a frontend and the code generator should be simple. The transformation will be simple if just the format of the interface alone were changed to the IDL format. However, the purpose behind the transformation is to make it easy to build new tools. It may be desirable that these frontends output all the useful information they can even if it can be re-computed. For example, the semantic analysis phases of the above compiler frontends should include in its symbol table not only the defining occurrences of symbols, but also all their uses. This latter information is computed during semantic analysis and would be useful to later optimization phases. The internal data structure of the semantic analyzer may contain this information, but its output data may not. Usually, the internal data structures of tools contain complex relationships between the data items, but the output data does not contain these relationships.

Consider a tool whose output data does not contain complex relationships between data items that are present in the internal data structure. These relationships

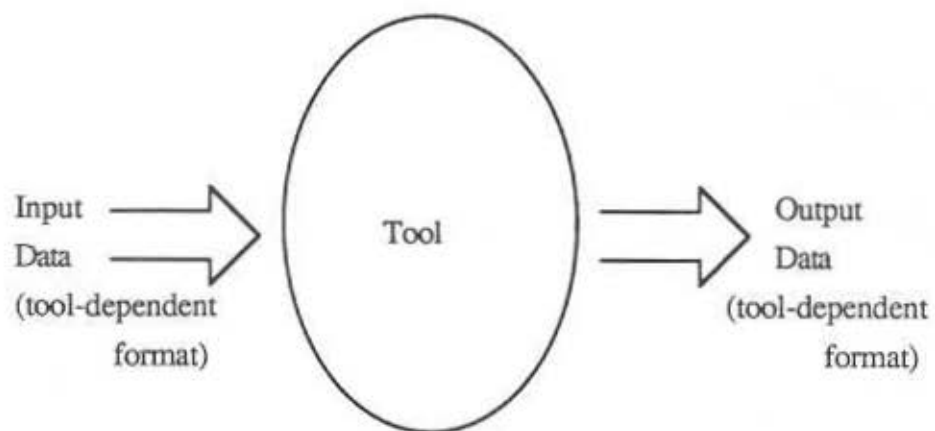


Figure 3.1: Example of a tool

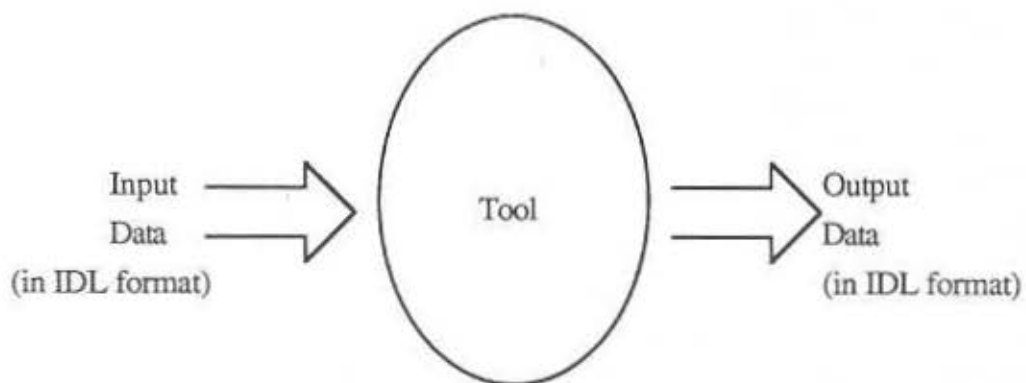


Figure 3.2: Example of a tool whose interfaces have been transformed

may be computed by other tools that use this tool's output as their input. If the output of this tool contains the complex relationships between the data items then these relationships need not be re-computed by every new tool that is implemented. Therefore it is desirable that the output of the tool contain the complex relationships between the data items.

Output formats of tools are restricted by the linear nature of the output interface. These formats are usually simple and do not describe complex relationships between data items that are present in the internal data structure. These relationships are computed internally after the input data is read and are ignored when output data is being written out. However, IDL can be used to describe complex relationships between data items. These relationships are embedded as attributes of the data items. The relationships are preserved when an IDL structure is externalized in the IDL format. If the interface routines were transformed to generate IDL instances, then these relationships can be output as well.

Since the IDL format is being used to exchange data, it is necessary that a tool also accept IDL instances as input. Input interfaces need to be transformed to accept IDL instances. Further, these transformations should not affect the tool's functionality. Special care is needed when input interfaces are being transformed.

To transform an interface, the methodology requires the tool and the interface that is to be transformed be known. The methodology is not concerned with specifying the IDL structure that should be associated with the interface being transformed. The methodology assumes that this IDL structure is completely specified. If the interface is an output interface then it is assumed that the information present in the output IDL instance is already computed by the tool. If the interface is an input interface then it is assumed that the input IDL instance contains all the information required by the tool's input. The method by which the output IDL instance may be computed from the information available in the tool (for output interfaces), or how the input IDL instance may be used to compute the tool's input data or internal data structure (for input interfaces) must also be known.

The methodology is applied as follows. There are many aspects to the problem of transforming an interface to communicate an IDL instance. Some of these aspects may be cast as dimensions. These dimensions when taken together characterize the set of problems and may be considered as the problem space. A particular problem can be examined and its characteristics determined. Using these characteristics, the particular problem may be identified with a point in the problem space. Associated with each point in the problem space is an approach applicable to that point along with the metrics of the transformation. The approach specifies how an interface may be transformed to communicate an IDL instance. Some points in the problem space have no approaches associated with them. The metrics associated with each approach discuss how easy the process of transformation would be. Sometimes more than one approach may be applicable to a point in the space. Then the metrics for all the approaches will be listed. The approach that is best suited to the problem should be chosen.

In the subsequent sections the dimensions, approaches and metrics are defined.

Following these definitions, a mapping of the points in the problem space into approaches is provided. Metrics are then provided for each solution. The chapter concludes by discussing the merits and deficiencies of this methodology.

3.1 Dimensions Of the Problem Space

The problem space has four dimensions. Some of the problems may be multi-valued along some of the dimensions. In that case, the different values have to be considered separately to determine the approach that is best suited.

Input or Output This dimension specifies if the interface that is being transformed is an input interface or an output interface.

Choice of Data A problem may have multiple values for this dimension. The IDL structure specifies the data structure that should be communicated by the interface. This data structure contains some complex relationships between the data items. For example, an attributed syntax tree output by the semantic analysis phase of a compiler contains, for every symbol, the defining occurrence of the symbol.

If an output interface is being transformed, then these relationships have to be computed. They may be computed from different forms of data. This dimension specifies the forms of data from which these relationships may be computed. Similarly, if an input interface is being transformed, these relationships should be translated to data meaningful to the tool. This data may be in different forms. Since the information present in the input IDL instance may be limited, it may be possible to translate this instance into some forms of the data. There many different forms of data, but only the following are of interest.

External The choice of data is the external data. External data refers to data that is external to the original tool, i.e. its original input or original output data. If an output interface is being transformed, then this means that the original output data contains sufficient information to compute the output IDL instance. If an input interface is being transformed, this means that the input IDL instance contains sufficient information to compute original input data.

Consider an expression evaluator that accepts expressions in postfix form as input. If the input IDL instance is an expression tree, then the choice of data is external, since the expression tree can be converted into the postfix form.

Consider a tool that produces as output an expression in prefix form. If the output IDL instance is an expression tree, then the choice of data is external, since the expression tree in prefix form can be converted into the expression tree (the arity of the operators is known).

Internal The choice of the data is the internal data structure. This refers to data that is internal to the tool. In particular, this refers to the global data structure present in main memory during the execution of the tool. If an output interface is being transformed, this means that the state of the internal data structure (just before data output) contains sufficient information to compute the IDL instance. If an input interface is being transformed, this means that the input IDL instance contains sufficient information to compute the state of the internal data structure (just after the input data is read).

Consider a program that evaluates the type of an expression. This program produces as output the expression tree along with the type of the expression. If the IDL structure requires the type of all sub-expressions of the expression, then that choice of data is internal (i.e., not available in the tool's original output, but computed as a side effect in the internal data structure).

Consider a program that evaluates an arithmetic expression. This program accepts as input an infix expression. The internal data structure stores the expression tree. If the input IDL instance is the expression tree, then the choice of data is internal, since the internal data structure can be computed from the input IDL instance.

If the tool's only input interface is being transformed, the input IDL instance contains sufficient information to compute the state of the internal data structure and the original input data. That is, the problem of transforming an input interface is always multivalued in this dimension. However, the value the particular problem has along the dimension 'Complexity of Computation' is dependent on the choice of data.

Complexity of Computation The preceding dimension concerns a computation that needs to be performed, that of adding relationships not explicit in the tool's original input or output. This dimension considers the complexity of that computation. It is not possible to give a precise definition of complexity. Characterizations of *simple* computation are

1. The computation is simple if there is correspondence between attributes in the IDL structure and portions of the data.
2. The computation is simple if two attributes that are independent in the IDL instance are also independent in the data.

Two attributes in an IDL instance are independent if their values which are directed graphs do not have any common nodes. Independence in the data is defined analogously. A *complex* computation is one that is not simple. For example, the parser of a compiler may output the parse-tree. If the IDL structure required that every symbol in the parse tree point to its declaration, that would be a complex computation.

Constraints on the Implementation There are many attributes of the problem that are pertinent to this dimension. A problem may have multiple values along this dimension. The different values along this dimension are as listed below.

1. Availability of the external data format.
2. Availability of the name space used by the program.
3. Documentation about the internal data structure.
4. Availability of the source code, with documentation.
5. Availability of the source code for modification.

3.2 Approaches

There are many ways in which an interface may be transformed to read or write instances in the IDL format. This section discusses some of the general approaches. The approaches that apply to a few specific cases are not presented. Some of these approaches can be cast into one of the general approaches after some modifications to the problem. These techniques are described in Section 3.6.

When an interface is being transformed the IDL data structure that should be communicated across the interface must be specified. An *external IDL instance* is an instance of this IDL data structure that is in the IDL format. An *internal IDL instance* is an instance of an IDL data structure that is stored in a main memory data structure. This data structure may be a derivation of the IDL structure specification that is transmitted across the interface. The extra attributes available in the internal IDL instance need to be computed and may be used ease the transformation of the interface. Routines provided by IDLC can be used to convert between the internal and external IDL instances. There are 10 approaches and they fall into two categories.

Converting between IDL instance and external data These approaches employ filters to convert between the IDL instance and the external data of the tool. A *filter* is a program that connects the interface to the external world by converting the data from one format to another. Examples of these approaches are illustrated in the Figure 3.3. These approaches are non-invasive, in that the tool is not modified in any way. Only the working environment is changed. These approaches require a complete understanding of the external data format of the tool.

There are four such approaches. The first two apply to input interfaces and the next two apply to output interfaces.

1. The tool's input is created from the external IDL instance. The filter reads the input IDL instance incrementally and processes it to produce the original input of the tool. Small parts of the input IDL instance are processed to produce portions of the original input data of the tool. These portions are produced in the order in which the parts are read. Routines provided by IDLC cannot be used to read the input IDL instance incrementally.

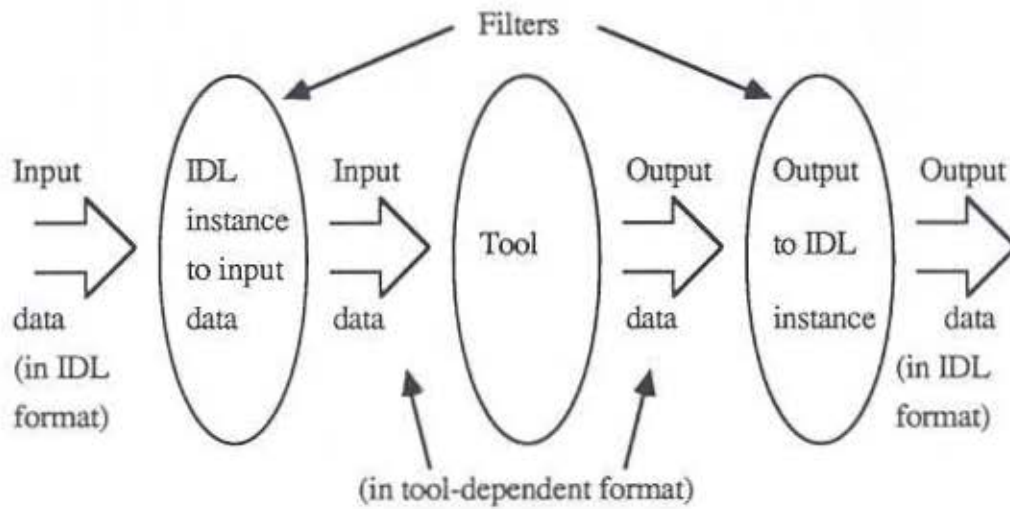


Figure 3.3: Examples of filters

Consider an expression evaluator that requires the expression tree encoded in preorder. The input IDL instance contains the expression tree, which is encoded in preorder. For example, the expression

```
add_exp[
  left_exp sub_exp[
    left_exp constant[value 2]
    right_exp constant[value 3]]
  right_exp constant[value 5]]
```

may be transformed into

```
+
  -
    2
    3
  5
```

by a simple scanner that transforms

```
'add_exp['      → '+'
'sub_exp['      → '-'
'left_exp'      → ''
'right_exp'     → ''
']'             → ''
'constant[value' r']' → x
```

where x is an integer.

2. The tool's input data is generated from the internal IDL instance. The input IDL instance is read into main memory by routines provided by IDLC. The original input data required by the tool is computed from the IDL instance and is written out by the filter. Since the IDL instance is in memory, the attributes of the IDL instance can be examined in any order. It may sometimes be necessary to build other data structures to compute the original input data. Usually, the original input data should be computable from the IDL instance in memory and a few variables.

Consider an expression evaluator that expects the expression tree encoded in postorder. The input IDL instance contains the expression tree, but it is encoded in preorder. The expression tree can be read into main memory using routines provided by IDLC. The tree can then be traversed in postorder and the original input data constructed for the original tool.

3. The external IDL instance is created from the tool's output. This approach is similar to the approach 1 listed above. The filter reads the original output of the tool incrementally and processes it to produce the output IDL instance. Small parts of the tool's output are processed to produce portions of the output IDL instance. These portions are produced in the order in which the parts are read. Routines provided by IDLC cannot be used to output the IDL instance incrementally. These routines need to generate not only the values of the attributes, but also the names of the attributes. If the attribute names are not available in the output data, then it must be computed from the IDL structure specification.

Consider a tool that produces as output an expression in prefix notation, i.e. similar to the original input of Approach 1. The output IDL instance required as output is one that contains the expression tree. Since a tree in the IDL format is encoded in preorder, the IDL instance can be created from the output of the tool (the arity of the operators is known). The tree may be built depth first as follows. A stack of operators, initialized to empty, contains the count of the operands associated with each operator. When an operator (say '+') is encountered in the input, the operators on the top of the stack that have the count of the operands equal to their arity are removed. For each operator removed, a '[' is output. The count of operands of the operator on top of the stack is incremented. The attribute name corresponding to the number of the operand ('left_exp' if 1, 'right_exp' if 2) is output. The current operator is added to the stack and its count of operators is set to 0. The name of the current operator (i.e. 'add_exp[') is output. If a constant (say x) is encountered then 'constant[value x]' is output. The stack has to be set up initially and special processing is required on end of input.

4. The internal IDL instance is created from the tool's output. This approach is similar to the Approach 2 listed above. The output data of the tool is read and the IDL instance is created in memory as the data is being read. This IDL instance may then be output using write routines provided by

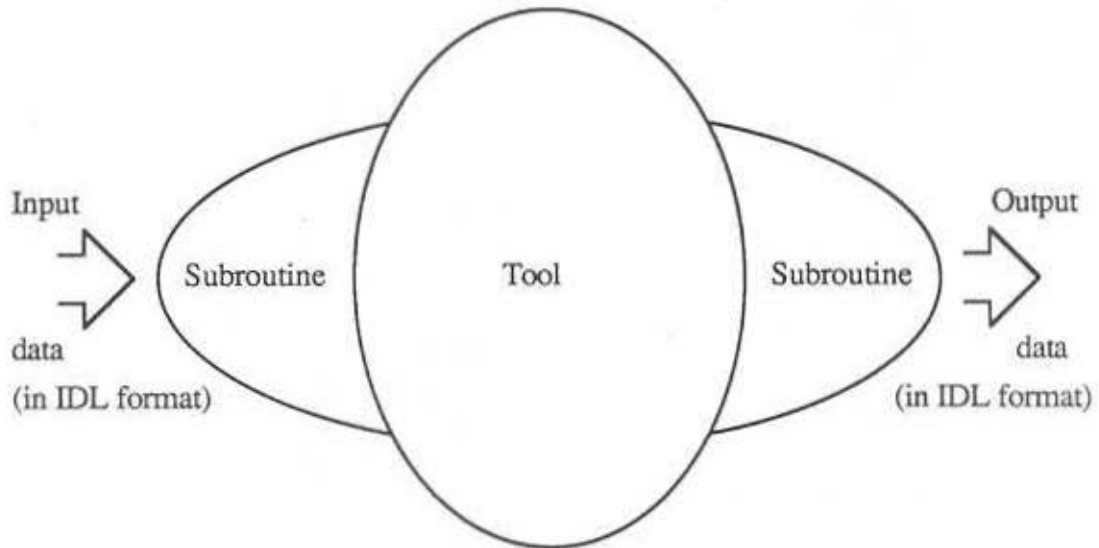


Figure 3.4: Use of subroutines

IDLC. Since the IDL instance is in memory the attributes of the IDL instance can be computed in any order. Sometimes intermediate data structures may have to be built to compute the IDL instance.

Consider an expression translator that takes as input an infix expression and produces as output a postfix expression. This output data needs to be converted into an IDL instance of the expression tree. This tree may be built bottom up as follows. A stack initialized to empty, contains the current operands. Whenever an operand (a constant) is encountered in the original output it is added to the stack. When an operator is encountered in the original output, the top two operands are removed from the stack, a new operand (which is the operator acting on the operands) is added to the stack. The only element on the stack at the end of the input is the required tree. This may then be output using routines provided by IDLC.

Converting between the IDL instance and internal data structure These approaches convert between the IDL instance and internal data structure of the tool. Usually, this can be done by a subroutine that converts between the internal data structure and the IDL instance as illustrated in the Figure 3.4. Approaches 7 and 10 replace whole or part of the internal data structure of the tool. These approaches require a complete understanding of the internal data structure of the tool. Further, the source code for the relevant portion of the tool should be available for modification. The tool must also be modified so that it does not read or write through its original interface.

There are six such approaches. The first three deal with transformation of input

interfaces and the rest with the transformation of output interfaces.

5. The tool's internal data structure is created from the external IDL instance. The subroutine reads the input IDL instance incrementally and builds the internal data structure of the tool. As each part of the input IDL instance is read, it builds or modifies a portion of the internal data structure. This is done in the order in which the parts of the input IDL instance are read. Routines provided by IDLC cannot be used to read the IDL instance incrementally.

The internal data structure of an expression evaluator contains an expression tree. The input IDL instance contains the expression tree encoded in the IDL format. The input interface may be transformed to accept this input IDL instance and build the internal data structure as suggested by this approach. The transformation can be simplified by considering some functions. The function *ascend()* sets the current node (a global variable) to its parent. The function *create(node)* creates the new node, adds the node as the left son of the current node if free, else as the right son, then sets the current node to the new node. The function *add(const)* adds the constant to left son of the current node if free, else as the right son. The current node is initialized to a special node called root that has only one son. The example given for Approach 1 may then be transformed by the following set of pattern matching rules.

'add_exp['	→	<i>create(plus_node)</i>
'sub_exp['	→	<i>create(minus_node)</i>
'left_exp'	→	do nothing
'right_exp'	→	do nothing
'constant[value' x']'	→	<i>add(x)</i>
']'	→	<i>ascend()</i>

6. The tool's internal data structure is created from the internal IDL instance. The subroutine reads the input IDL instance into memory using routines provided by IDLC. The internal data structure is computed by traversing the IDL instance in memory. Since the IDL instance is in memory, the attributes of the IDL instance may be examined in any order.
7. In this approach, the IDL input instance is read into an IDL structure using routines provided by IDLC. This IDL structure is a derivation of the input interface's IDL specification. This internal IDL structure replaces all or part of the internal data structure of the tool. A part of the data structure may still need to be built from the IDL data structure. All accesses and modifications to the internal data structure are replaced by equivalent accesses and modifications that act on both the internal data structure and the IDL data structure.

For example, a tool's input may be a list of items. The tool's internal data structure is also a list which is implemented using an array. This internal

data structure may be replaced by an IDL data structure. The access routines that accessed the original data structure have to be modified to access the new data structure.

8. The external IDL instance is produced from the internal data structure. This approach is similar to Approach 5. The subroutine writes out portions of the output IDL instance by interpreting portions of the internal data structure of the tool. Routines provided by IDLC cannot be used to output IDL instances incrementally. The external format requires the attribute names and their values. If the attribute names are not available in the internal data structure of the tool, then these names have to be hard-coded into the subroutine or encoded in a table. These attribute names may be obtained from the IDL structure specification. The internal data structure can be traversed in any sequence and the output IDL instance produced. An expression evaluator produces the value of the expression as its output, but its internal data structure contains the expression tree. If the output IDL instance is an expression tree, this can be done by doing a preorder traversal of internal data structure and outputting the operators and operands in the IDL format.

9. The internal IDL instance is produced from the internal data structure. This approach is similar to Approach 6. The IDL instance in memory is created from the internal data structure of the tool. The IDL instance may then be output using write routines provided by IDLC. Since the IDL instance is being built in memory, the attributes of the IDL instance may be computed in any order.

The internal data structure of the semantic analyzer phase of a compiler contains the attributed parse tree. This tree needs to be output in the IDL format. A routine that walks the parse-tree and builds the corresponding instance of the IDL structure in memory can be written. This IDL instance in memory can then be output by routines provided by IDLC.

10. In this approach an IDL data structure is built that may replace part or whole of the internal data structure of the tool. This IDL data structure is a derivation of the IDL specification of the output interface being transformed. The output IDL instance is written out using routines provided by IDLC. Like approach 7, macros that access and build the internal data structure of the tool have to be replaced by equivalent accesses and modifications to the internal data structure and the IDL data structure. For example, consider the semantic analyzer phase of a compiler. The output interface that outputs the errors detected in semantic analysis needs to be transformed to output IDL instances. This may be done by building an IDL data structure that is built as and when an error is encountered. Since errors are not very infrequent, this may be more efficient than computing the list of errors after the semantic analysis phase is completed.

Instead of transforming an existing output interface, a new interface to output

IDL instances can be created, by applying one of Approaches 8, 9 or 10. It is possible to build this interface without any side-effects on the tool. This offers the advantage that both the original output and the output IDL instance are available, so either output may be used.

3.3 Metrics

This section considers the metrics that may be used to evaluate the different solutions to transforming an interface. These metrics apply to the process of transformation and the particular tool whose interface has been transformed. In this section, a transformed tool is a tool whose interface has been transformed to communicate IDL instances. There are three kinds of metrics that are considered.

Metrics on the transformation These metrics are concerned with the cost of transforming the interface.

1. Length of time required to do the transformation
2. Manpower required to do the transformation.
3. Other resources that are required, e.g., disk space.

Metrics on the transformed tool These metrics are concerned with the extra cost incurred by using the transformed tool instead of the original tool.

1. Amount of extra space required (static and dynamic).
2. Amount of extra time required.
3. Maintenance cost of the tool.
4. Other resources required (disk space etc.,)

Robustness of the transformed tool These metrics are concerned with the robustness of the transformed tool to change.

1. Robustness over different versions of the tool.
2. Robustness over different versions of IDL and the IDL Toolkit.
3. Robustness over different versions of the environment.

3.4 Mapping

This section presents the mapping between the points in the problem space to approaches. There are four dimensions in the problem space: Input or Output, Data, Complexity of Computation, Constraints on the Implementation. Some problems may be multi-valued along some of the dimensions. Each of the different values have to be considered separately as though they were separate problems and the approach suitable found. From these approaches, the best approach should be chosen.

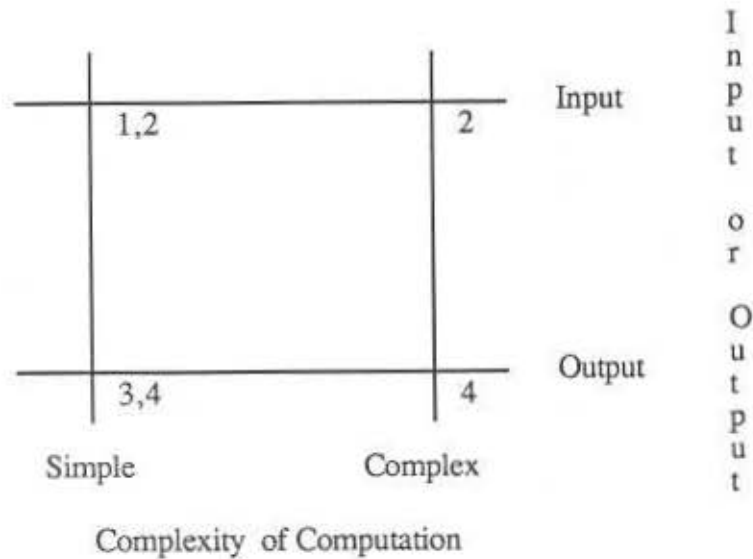


Figure 3.5: Approaches for problems with data dimension 'external'

The mapping is done as follows. The problem space may be partitioned into two along the data dimension. The problems in the first partition (i.e. data dimension 'external') may be solved by approaches in the first category. The problems in the second partition may be solved by the approaches in the second category. The dimension 'Constraints on the Implementation' refers to some conditions that have to be satisfied before the approaches may be applied. The other two dimensions determine the particular approach in the category that is applicable to the problem.

The approaches in the first category are suited to the problems that have the value along the data dimension as 'external'. For these approaches, the format of the original data must be known to transform the interface. The approaches suitable for the different problems are illustrated in Figure 3.5. The numbers identify the relevant approach.

The approaches in the second category are suited to problems that have the value along the data dimension as 'internal'. For these approaches the source code of the original tool must be available for modification. The approaches suitable for the different problems are illustrated in Figure 3.6. The numbers here also identify the relevant approach.

Approaches 1 and 5 suggest that the input IDL instance be processed incrementally. It is not clear how the IDL instance should be split up to be processed incrementally, or if such a split is possible at all. However, these two approaches are applicable only if the 'Complexity Of Computation' is simple. One characterization given of a simple computation is that two attributes that are independent in the IDL instance are also independent in the data. The IDL instance can be split into attributes that are independent and these can be used to generate pieces of the data

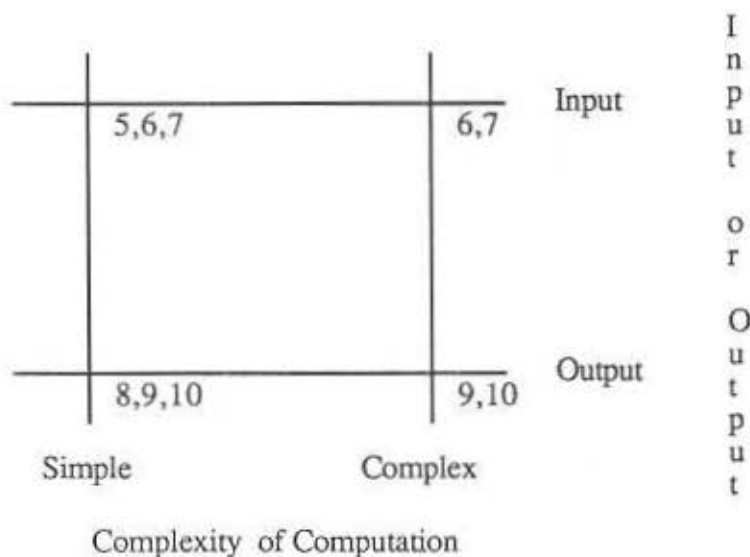


Figure 3.6: Approaches for problems with data dimension 'internal'

(either the original input data or internal data structure).

Similarly, Approaches 3 and 8 suggest that the data (either input data or internal data structure) be processed incrementally. These approaches are applicable only if the 'Complexity Of Computation' is simple. Another characterization given of a simple computation is that there is a correspondence between attributes in the IDL structure and portions of the data. These portions of the data should be used to compute the corresponding attributes of the IDL structure.

3.5 Metrics on the Solutions

This section considers the metrics on the solutions presented in the previous section. First the effect of modifications to IDL and IDL toolkit on the different approaches is examined. Then some aspects affecting more than one approach are presented. For each approach, a general discussion about the difficulty of the approach, followed by values for the different metrics, is presented.

The format in which the IDL data structures are written out (the IDL format) is fixed by the IDL toolkit. IDLC maps a structure specification in IDL into data structures in the target language and provides routines to read and write these data structures. IDLC also provides some operators to operate on these data structures. Changes to IDLC could change one or more of the above.

Approaches 1,3,5 and 8 do not use IDLC and the IDL toolkit but for the external representation format of an IDL instance. They are therefore robust over changes in IDLC with respect to the interface routines, data structures and operations on these data structures, provided by IDLC.

Approaches 2, 4, 6, 7, 9 and 10 use the interface routines and the data structure declarations provided by IDLC. Changes to the interface routines and the external format should not pose a problem, since the interface routines should still be able to read/write the data structures in the appropriate format. These approaches may need to be rewritten if there are changes to the data structures and the operators. These changes should be fairly simple since the basic nature of IDL will not change. It may be possible to automate the process of rewriting these approaches.

The changes to IDLC and the external format may affect the metrics on the transformed tool. For instance, a compact external representation may improve the performance of the transformed tool substantially. These changes may also affect the approach that is most suitable for a particular problem. These changes affect the "Complexity of Computation" of the problem. Other approaches may then become more suitable. For the example in Approach 1, if the input IDL instance were encoded in binary format [Biyani 1987], it may not be practical to build a scanner as shown.

Among the different approaches, those that create the IDL instance in memory (2, 4, 6, 7, 9 and 10) would require extra storage space at run-time. However if the IDL instance is in memory, it is possible to access the attributes in any order. A disadvantage in building IDL instances of complex structures in memory is that accessing them requires significantly more code.

An interface is transformed by approaches 1, 2, 3 and 4 by the use of a filter. These filters can be debugged separately, before they are coupled with the tool. Hence, they can be developed faster. These filters may be constructed using tools like SED, AWK and LEX [Kernighan & Pike 1984]. Then, the time for transforming the interface will be fairly short.

Approaches 5 and 6 are not generally attractive for transforming input interfaces. The incorrect computation of the internal data structure could alter the functionality of the tool. Even though the same problem exists when the tool's original input is computed, the tool may be better equipped to handle inconsistencies in its original input data rather than the internal data structure. Approaches 5, 6, 7, 8, 9 and 10 are attractive, if the tool has simple mechanisms to access the internal data structure. Table-driven data structures tend to have simple access mechanisms.

From the mapping in the previous section it appears that the values for the 'Constraints on Implementation' dimension do not affect the choice of the approach. However, these values affect the metrics on the solution. For example, approaches 5, 6, 7, 8, 9 and 10 require just that the source code be available for modification. While this is sufficient, the availability of the documentation about the internal data structure would reduce the time and effort required to transform the interface. The source code may itself be considered as documentation about the internal data structure, but its quality is likely to be poor. If more constraints are satisfied then the transformation using approaches 5, 6, 7, 8, 9 and 10 is easier. For approaches 1 to 4 the availability of the external data format is sufficient and the other constraints do not change the metrics on the transformation.

Approach 1 The input IDL instance is processed incrementally by splitting it into small parts and the equivalent portion of the original input data generated.

The characteristics of the parts determine the difficulty of transforming the input interface. The size of each part determines the difficulty in processing each part to produce the corresponding output. The number of different parts determines the amount of code that needs to be written. However, some of the processing may be common between the different parts and the code may be shared. Another aspect to consider is the difficulty in developing a reader to read the different parts of input IDL instance. This approach is appropriate when the input IDL instance can be split into a few different parts, whose processing is simple.

Therefore, the transformation should be simple, and hence the length of time and the manpower required should be small. The metrics on the transformed tool are given below.

1. The program should be small, since simple data is being processed. Since no large data structures are being stored, the amount of dynamic data space required is also small.
2. Processing the input IDL instance to produce the original input data to the tool is simple and efficient in time. Since the IDL format is space inefficient, I/O might make the filter slow, especially for a large input IDL instance.
3. Maintenance cost of the tool is not significantly altered. There is however a small additional cost in maintaining the filter.
4. The output of the filter has to be connected to the original input of the tool. On the UNIX operating system, this can be done easily using *pipes* [Kernighan & Pike 1984]. On another operating systems, an intermediate file may have to be created.
5. The transformed tool is robust over changes in the original tool in that the filter can be used with different versions of the original tool as long as the input format does not change.
6. Since the transformed tool depends only on the external format of input IDL instance (which is unlikely to change), it is robust over changes in IDL and the IDL Toolkit. However, if the processing depends on the order of the different attributes, then it is less robust over changes in IDL, the IDL Toolkit or changes in the IDL specification.
7. The tool does not depend on any special features of the environment, therefore it is robust over changes in the environment.

Approach 2 The input IDL instance is read into memory by routines provided by IDLC. The original input data is computed by traversing the IDL instance in memory. Since the read routines are provided by IDLC there is no effort in writing these routines. The difficulty of the transformation depends on the complexity of the computation that computes the original input data.

Since this approach would be used for moderately complex transformations, the length of time and manpower should be moderate. The metrics on the transformed tool are given below.

1. For moderate sized IDL structure specifications, the program would be large because of the readers provided by IDLC. Since the IDL instance is stored in memory, dynamic data space required is large.
2. For large input IDL instance I/O might slow down the tool. This effect may dominate the time taken to compute the original input data.
3. There is the added maintenance cost of maintaining the filter.
4. As in the previous approach, a mechanism is required to feed the output of the filter as the input of the tool.
5. The transformed tool is robust over changes in the tool as in Approach 1.
6. The transformed tool is robust over changes in IDL and the IDL Toolkit since routines provided by IDLC are being used to read input IDL instance.
7. The transformed tool is robust over changes in the environment because of reasons outlined in the previous approach.

Approach 3 This approach is very similar to Approach 1 listed above. The only difference is that Approach 1 considers the transformation of an input interface, while this approach refers to the transformation of an output interface. The original output data is processed incrementally by splitting it into small parts and the equivalent portion of output IDL instance is generated. Like Approach 1 the size and number of parts decide the difficulty and the size of the code. A writer has to be developed that writes out the output IDL instance. This approach is appropriate when the original output data of the tool can be split into a few different parts, whose processing is simple.

As explained in Approach 1, the length of time and manpower should be small. The metrics on the transformed tool are given below

1. Like Approach 1, the static and dynamic data requirements should be small.
2. Like Approach 1, the filter may be slowed by I/O.
3. Like Approach 1, the maintenance cost is not significantly altered.
4. Like Approach 1, a mechanism is required to supply the output data of the tool as the input data of the filter.
5. The transformed tool is robust over changes in the tool in that the filter can be used with the different versions of the tool as long as the output format of the tool does not change.
6. Since the tool depends only on the IDL format, it is robust to changes in IDL and the IDL Toolkit.

7. The tool is robust over changes in the environment.

Approach 4 The in-memory IDL data structure is created and this is output using routines provided by IDLC. The original output data is read and as it is read the in-memory IDL data structure is constructed. Since the routines to produce the output IDL instance are generated by IDLC there is no effort in writing these routines. The difficulty of the transformation depends on the complexity of the computation that computes the internal IDL instance.

As in Approach 2 listed above, the length of time and manpower should be moderate. The metrics on the transformed tool are the same as for Approach 2.

Approach 5 The input IDL instance is processed incrementally by splitting it into small parts and an equivalent portion of the internal data structure is built. The difference between this approach and Approach 1 is that the input IDL instance builds the internal data structure, rather than the original input data. The metrics are also similar. This approach is appropriate when the input IDL instance can be split into a few different parts, whose effect on the internal data structure is simple.

The transformation of the input interface should be simple. The length of time and the manpower required should be small. The metrics on the transformed tool are given below.

1. The subroutine should be small, since simple data is being processed. Since large data structures are not stored, the amount of dynamic data space required is also small.
2. Processing the input IDL instance to build the internal data structure is simple and efficient. Since the IDL format is space inefficient, the transformed tool may be slowed down because of I/O.
3. The maintenance cost of the tool is higher in this approach than in the first four approaches. Since the tool is being modified, bugs in the modification are more difficult to identify and correct. Since it is difficult to detect inconsistencies in the internal data structure (which is built by this approach), the maintenance costs are high.
4. No extra resources are required.
5. The transformed tool is not robust over changes in the tool. Some internal modifications in the tool may require that the interface be transformed again, not just re-implementation of the transformation, since other approaches may be better.
6. As in Approach 1 the transformed tool is robust over changes in IDL and the IDL Toolkit, provided the processing is independent of the order of the different attributes in the input IDL instance.
7. The tool is robust over changes in the environment.

Approach 6 The input IDL instance is read into memory by routines provided by IDLC. This approach differs from Approach 2 in that the internal data structure is computed by this approach, rather than the original input data. The internal data structure of the tool is computed by traversing the IDL instance in memory. The read routines are provided by IDLC. The difficulty of the transformation depends on the complexity of building the internal data structure.

Like Approach 2, the length of time and manpower required for the transformation will be moderate. The metrics on the transformed tool are given below.

1. Since the internal data structure of the tool and the input IDL instance are being stored in memory concurrently, the dynamic storage space requirement larger than Approach 2. The static storage space requirement will also be large because of readers provided by IDLC.
2. Like Approach 2 the increase in I/O time will be the major contributor to the loss in speed. However, since the original input data is not written out and read again, the decrease in execution speed will be less than that of Approach 2.
3. Like Approach 5 the maintenance cost of the transformed tool is higher than in the first four approaches.
4. No extra resources are required.
5. Like Approach 5 the transformed tool is not robust over changes in the tool.
6. Like Approach 2 the transformed tool is robust over changes in IDL and the IDL Toolkit.
7. The tool is robust over changes in the environment.

Approach 7 The input IDL instance is read into an IDL structure. This structure is a derivation of the input interface's IDL specification. Some portion of the internal data structure of the tool may need to be computed. The accesses to the internal data structure need to be replaced with equivalent access routines that act on both the internal data structure and the IDL structure. The complexity of these determine the complexity of the transformation.

If the internal data structure were to be replaced by the internal IDL data structure, then all accesses to the internal data structure need to be changed to equivalent accesses to the internal IDL data structure. But the original internal data structure need not be computed. If only a part of the internal data structure were being replaced by the IDL structure, then the remainder has to be computed from the input IDL instance, but only some of the accesses to the internal data structure need be changed.

To replace internal data structures that are not simple, a substantial portion of the tool's source code may have to be rewritten. Therefore a moderate amount of manpower and time would be required. The metrics on the transformed tool are given below.

1. Like Approach 2, the static and dynamic storage requirements will be large. Unlike Approach 6, only a part of the internal data structure may need to be stored. But, the IDL structure may require more space, since it may contain extra attributes.
2. Like Approach 6 increase in I/O time may affect the speed of execution. The processing time, even though increased may not be a major factor.
3. The maintenance cost of the tool is fairly high, even higher than that of Approach 6, since extensive modification is required.
4. No extra resources are required.
5. Like Approach 5 the transformed tool is not robust over changes in the tool.
6. Like Approach 2 the transformed tool is robust over changes in IDL and the IDL Toolkit.
7. The tool is robust over changes in the environment.

Approach 8 The output IDL instance is produced by traversing the internal data structure. A writer writes out the IDL instance in the IDL format. This approach is suitable if the internal data structure contains all the attributes required by the IDL structure specification. The complexity of the traversal required to generate the output IDL instance determines the complexity of the transformation.

The traversal of the data structure should be simple. Therefore, the length of time and the manpower required should be small. The metrics on the transformed tool are given below.

1. Like Approach 5 the static and dynamic space required should be small.
2. Like Approach 5 the tool may be slowed down by I/O.
3. Like Approach 5 the maintenance cost of the tool is higher for this approach than for the first four approaches.
4. No extra resources are required.
5. As in Approach 5, the transformed tool is not robust over changes in the tool.
6. As in Approach 3, the transformed tool is robust over changes in IDL and the IDL Toolkit.
7. The tool is robust over changes in the environment.

Approach 9 The IDL instance is created in memory and this is output using routines provided by IDLC. This approach differs from Approach 4 in that the in-memory data structure is created from the internal data structure, rather than the original output data. The IDL instance in memory is created by traversing the internal data structure. The difficulty of the transformation depends on the complexity of the computation to compute the IDL instance in memory.

As in Approach 2, the length of time and manpower should be moderate. The metrics of the performance and robustness of the transformed tool are the same as for Approach 6.

Approach 10 The internal data structure of the tool is replaced in whole or in part by an IDL data structure that is a derivation of the IDL specification for the output interface being transformed. Like approach 7 the accesses to the internal data structure have to be modified so that they act on both the internal data structure and the IDL data structure. The new internal data structure (IDL structure and a part of the original internal data structure) is directly computed from the input data, since the accesses have been modified.

Like Approach 7, the transformation of the interface should be fairly complex. Therefore a moderate amount of time and manpower would be required. The other metrics are the same as those for Approach 7.

3.6 Other Techniques

The methodology considers only two points along the data dimension, i.e. external data and internal data structure. When an output interface is being transformed, the internal data structure of the tool or the output data may not contain adequate information to compute the output IDL instance. The methodology that is presented above will not be able to solve such a problem. This section presents techniques that may be used to solve some of those problems. Each of these techniques modifies the problem so that the methodology can be applied to it.

Technique 1 This technique may be used in the transformation of an output interface. A tool may compute some attributes that are required by the IDL instance, but these attributes may be discarded and not stored in the internal data structure. Instead, a special data structure may be built to store such attributes. The new internal data structure can be considered to be the special data structure and the internal data structure of the original tool. One of Approaches 5, 6, 7, 8, 9 and 10 may be applicable. For example, the semantic analysis phase of a compiler may compute the usage of the symbols, but may not store it in its internal data structure. This information can be captured in a special data structure and then used to help in the transformation. The transformation is a little more complex, since the information is present in two different structures. The transformed tool requires more static and dynamic data space, but not substantially so.

Technique 2 This technique may be used in the transformation of an output interface. A tool may compute its output incrementally. The internal data structure of the tool may only contain information about a portion of the output. A new data structure can be built that accumulates the internal data structure. This new data structure is updated whenever data is input. This new data structure

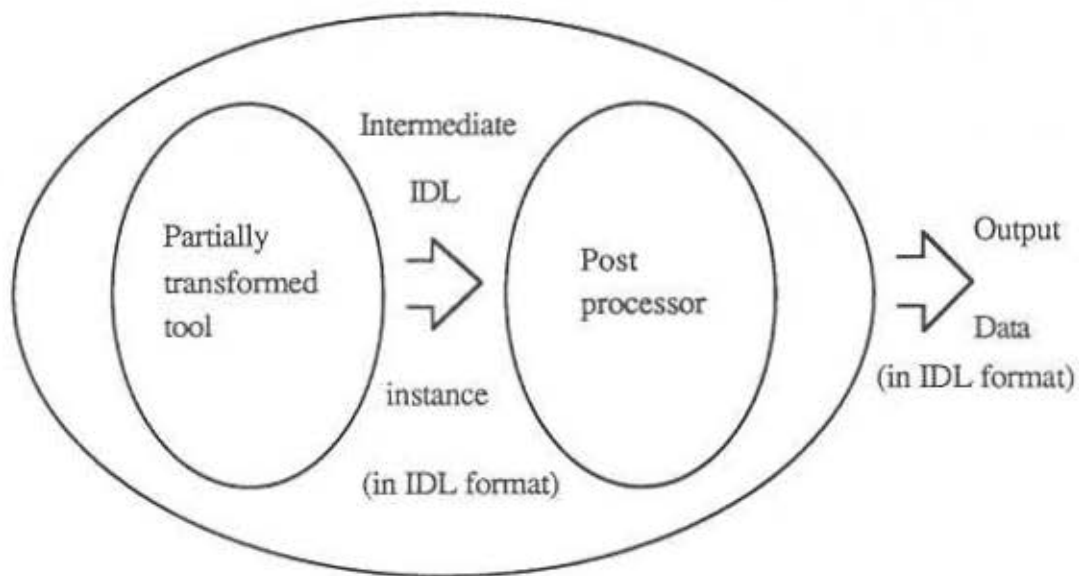


Figure 3.7: A two step transformation

serves as the internal data structure for applying the methodology. One of Approaches 5, 6, 7, 8, 9 and 10 may be applicable. The transformed tool requires more static and dynamic data space.

Technique 3 This technique may be used in the transformation of an output interface. A tool may not output sufficient information in its output. The tool can be modified so that the extra attributes are also output. The new output now contains sufficient information to perform a transformation. One of Approaches 1, 2, 3 and 4 may be applicable.

Technique 4 This technique may be used to transform an input or an output interface. It may be far too complex to perform the transformation of the interface in one step. An intermediate IDL structure may be developed that can be used to simplify the transformation. This technique is illustrated in Figure 3.7. This technique may slow down the transformed tool considerably. Since a number of different programs are involved, significant additional static and dynamic data space may be required.

3.7 Transforming Many Interfaces

The methodology may be used to transform one interface at a time. If more than one interface is being transformed, the methodology suggests that they be transformed independently. However when multiple interfaces are transformed, a sequential approach may not result in the best overall solution. Approaches 7 and 10 build the IDL

data structures in memory. If an input interface were transformed using Approach 7 and an output interface using Approach 9, then these transformations could be combined in such a way that the total cost would be less than the sum of the cost of transforming the two interfaces.

For example, the internal data structure of an expression type checker may contain an expression tree, but the checker may input and output expressions in infix format. If the input and output interface need to be transformed to input and output expression trees in IDL format, then the internal data structure can be replaced by an IDL data structure which is a derivation of the input and output data structures. The input and output IDL instances can then be read and written using routines provided by IDLC. Replacing the internal data structure with the IDL structure may be easier than transforming both the input and output interfaces separately.

3.8 Conclusions

This chapter presents a methodology for transforming an interface. First the particular problem of transforming an interface is examined and its characteristics determined. A set of approaches that may be used to transform an interface are considered. A mapping associates the characteristics with an appropriate approach. The metrics associated with the different approaches are discussed. Other techniques that may be used to help the transformation of an interface are evaluated. The chapter ends by considering the transformation of more than one interface. The methodology is summarized in Appendix D.

Chapter 4

Transforming a Syntax-directed Editor

This chapter discusses an application of the methodology presented in Chapter 3 to transform an input and an output interface of a syntax-directed editor. First, the tool whose interfaces are being transformed is discussed in detail. Then, the transformation of an output interface is considered. The methodology is then applied to transform an input interface.

A text editor is used to enter and modify text. A syntax-directed editor is used to create and modify programs in a particular language. The syntax of the language is the syntax of the data that may be edited by the editor. For example, a syntax-directed editor for the Pascal programming language may be used to create and edit programs in Pascal. The text produced by this editor is guaranteed to be syntactically correct. Such an editor can be generated using the Synthesizer Generator [Reps & Teitelbaum 1984].

The Synthesizer Generator is a tool that generates syntax-directed editors when provided with their specification. A part of this specification is the *abstract syntax* of the data. The abstract syntax is a set of grammar rules, essentially productions of a context free grammar [Hopcroft & Ullman 1979]. Each production has a name, known as the *operator*. The *abstract syntax tree* is an instance of the abstract syntax represented as a tree. A set of *parsing declarations* specify how the abstract syntax tree may be computed from text that is input to the editor. A set of *unparsing declarations* specify the textual representation of the abstract syntax tree. Terms and productions in the abstract syntax may have *attributes* associated with them. A set of *attribution rules* specify how the attributes may be evaluated from the other attributes in the tree. The *attributed syntax tree* is the abstract syntax tree along with these attributes.

Given these specifications, the Synthesizer Generator generates a syntax-directed editor that computes the abstract syntax tree from the input text using the parsing declarations, evaluates the attributes as specified by the attribution rules, and computes the textual representation of the abstract syntax tree as specified by the unparsing declarations. This textual representation may be edited interactively and the attributed syntax tree is recomputed whenever the text is changed. The text that

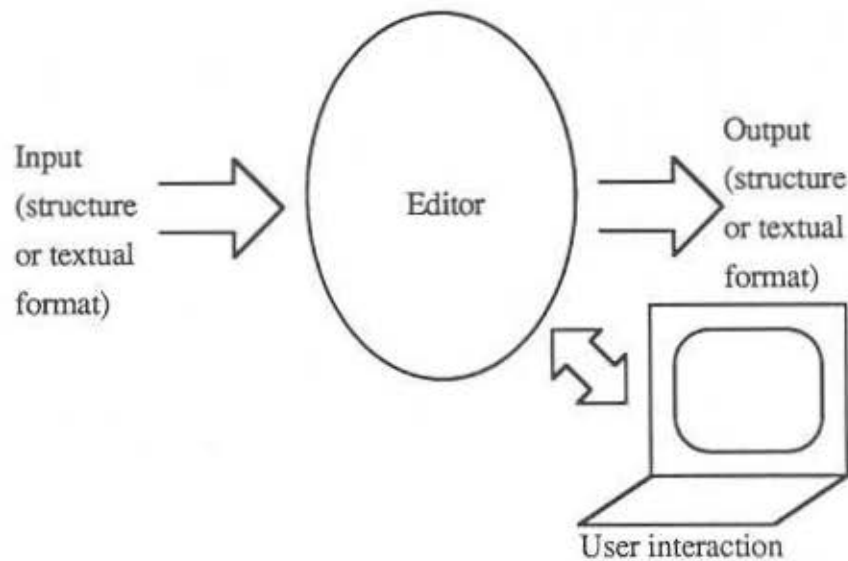


Figure 4.1: An editor generated using the Synthesizer Generator

is being edited is also modified to reflect the changes in the abstract syntax tree.

An editor generated using the Synthesizer Generator (see Figure 4.1) inputs and outputs data in two different formats. The first, the *textual format*, is the textual representation of the abstract syntax tree. This may be computed from the abstract syntax tree using the unparsing rules. The editor accepts as input, textual data and computes the abstract syntax tree by using the parsing rules. The second, the *structure format*, contains an abstract syntax tree encoded as a list of operators with their name and arity, followed by the operators in the abstract syntax tree listed in preorder.

The Synthesizer Generator can be used to build a number of fairly sophisticated editors that check both the syntax and semantics of input programs for programming languages like Pascal, C and Fortran77. The internal data structure of these editors contains the abstract syntax tree along with the attributes. However the output of these editors contains only the abstract syntax tree. The methodology presented in Chapter 3 will be used to transform such an editor to output an attributed tree. The input and output interfaces of an editor that accepts a simple expression language in infix format and functions as a desk calculator will be transformed. The transformed interfaces communicate their instances in the IDL format.

The data structure that is input and output by the transformed editor should be an instance of the attributed syntax tree represented in the IDL format. The attributes associated with the nodes in the abstract syntax tree that are computed by the editor will be present in the output. The editor should also accept its output as input. The transformation of an editor using the methodology demonstrates how an editor generated by the Synthesizer Generator may be transformed to input and output IDL

instances. An editor for the Pascal language, generated by the Synthesizer Generator can be transformed to output an attributed syntax tree. This attributed syntax tree can be used by a code generator to generate executable code. The syntax-directed editor thus transformed can be used as the frontend for a prototype compiler.

The methodology given in Chapter 3 may be used to transform only one interface at a time. Therefore the transformation of the output interface of the editor will be considered first. Then, the transformation of the input interface will be considered.

4.1 Transformation of the output interface

To apply the methodology the characteristics of the problem along the different dimensions of the problem space have to be determined. The problem space has four dimensions: Input or Output, Data, Complexity of Computation, Constraints on the Implementation. The output interface is being transformed, therefore the value along the 'Input or Output' dimension is 'Output'. Along the 'Constraints on the Implementation' dimension, the formats of data input and output by the tool (external data format) are available and the source code is available for modification. However, the name space used by the program, the documentation about the internal data structure and the documentation on the source code are not available. The 'Data' and the 'Complexity of Computation' dimensions are discussed below.

The problem of transforming the output interface of the desk calculator is multi-valued along the data dimension i.e., its value is both 'internal' and 'external'. The output IDL instance may be computed from the data output by the editor as well as the internal data structure of the editor.

The editor outputs its data in two formats, a textual format and a structure format, representing the abstract syntax tree. The attributed syntax tree, represented in IDL format, may be computed from either of these outputs. The syntax tree may be computed by parsing the text output by the editor. The syntax tree may also be built from the structure output of the editor. In either case, the attributes of the nodes in the syntax tree may be computed using the attribution rules. An attribute in the output IDL instance is independent of other attributes in the IDL instance. However, the evaluation of an attribute by applying the attribution rules on the abstract syntax tree may require the evaluation of the other attributes. This is one of the characteristics of a complex computation as defined in Chapter 3. Hence the 'Complexity of Computation' of both these computations (i.e. computing the output IDL instance from the textual or structure input) is 'Complex'.

The internal data structure of the editor contains an attributed tree that stores the abstract syntax tree of the data being edited along with the attributes. Each node in the attributed tree is an instance of a production in the abstract syntax. The node has as its sons instances of all the nonterminals that are part of the right hand side of the production. The node also has as its sons all the attributes of the nonterminal on the right hand side of the production and all the attributes local to the production. Some of these sons are subtrees and some of them are constants (e.g., integer). Further, there are number of arrays that contain properties of productions

and the nonterminals in the syntax. Two of them are, the number of nonterminals in each production, and the names of the nonterminals. There are also many flags that are used to ensure that the attributes of the syntax tree are computed in a optimal manner. Many macros are available to access the internal data structure of the tool easily. Since the attributed syntax tree is contained in the internal data structure, the attributes required by the IDL data structure do not have to be computed. The complexity of this computation is simple.

Summarizing, this problem maps to the following two points in the problem space. One of the points has along the data dimension, the value external and along the complexity of communication dimension, the value complex. The other point has along the data dimension the value internal and along the complexity of communication dimension the value simple. Both points have along the input or output dimension the value output. From the mapping given in Chapter 3, the Approaches 4, 8, 9 and 10 are applicable. The metrics on the transformation of the interface of the tool using the different approaches must be considered to choose the most appropriate approach from these approaches.

In Approach 4, the in-memory data structure is created from the output of the tool(either the structure or textual output) and this is output using routines provided by IDLC. The complexity of the computing the internal IDL instance determines the difficulty of the transformation. In the case of the desk calculator, computing the attributes of the different nodes in the syntax tree using the attribution rules can be moderately difficult, hence the transformation is moderately difficult. For an editor generated from a more complex language, computing the attributes of the different nodes is likely to be more difficult.

In Approach 8, a subroutine writes out the IDL instance in the IDL format by traversing the internal data structure. This IDL instance contains the attributes in preorder. The complexity of the traversal determines the difficulty of the transformation. In the case of the desk calculator, the internal data structure contains the attributed syntax tree which can be traversed in preorder easily. Therefore, the transformation is fairly simple. This holds even for an editor that is generated from a more complex language.

In Approach 9, the IDL instance is created in memory and this is output using routines provided by IDLC. The difficulty of the transformation is determined by the complexity of computing the IDL instance in memory. While it is fairly simple to compute the attributes of the different nodes of the tree, creating an internal IDL data structure is complicated, since a separate routine is required for each node type in the IDL specification. This is fairly complicated even for a small specification. For an editor generated from a more complex language, it would be more complicated.

In Approach 10, the part of the internal data structure that contains the attributed syntax tree is replaced by an IDL data structure that is a derivation of the IDL specification for the output interface. This part of the internal data structure other than containing just the attributed syntax also contains information about how the attributes may be evaluated efficiently. This information is stored in flags at every node of the abstract syntax tree. Since sufficient documentation is not available


```

exp : Null()
    | Sum(exp,exp)
    | Diff(exp,exp)
    | Prod(exp,exp)
    | Quot(exp,exp)
    | Const(INT)
    ;
exp { synthesized INT v;};

```

Figure 4.2: A part of the specification to the Synthesizer Generator

```

exp ::= Null | Sum | Diff | Prod | Quot | Const ;
exp => v : Integer;
Null => ;
Sum => Sum1:exp, Sum2:exp;
Diff => Diff1:exp, Diff2:exp;
Prod => Prod1:exp, Prod2:exp;
Quot => Quot1:exp, Quot2:exp;
Const => Const1:Integer;

```

Figure 4.3: IDL specification for the abstract syntax in Fig. 4.2

about the internal data structure, it is not clear how these flags may be evaluated. Further, all accesses to the attributed syntax tree have to be modified to access the IDL structure. Even though macros are used to access the internal data structure, these macros are highly parametrized (e.g. `set_attr(a,b,c)`, set attribute *b* of node *a* to *c*). Given the nature of the data declarations produced by IDLC, it would be difficult to provide such macros to access the IDL data structure. Therefore replacing the internal data structure with an IDL data structure is complex. The transformation using this approach is fairly complex.

Judging by the difficulty of the transformation, Approach 8 seems to be the best approach at hand. Instead of transforming an existing interface, a new interface was created to output the IDL instance. The transformed editor can output the attributed syntax tree in IDL format and the abstract syntax tree in text and structure formats.

The portion of the specification used by the Synthesizer Generator to generate the desk calculator is shown in Figure 4.2 and the corresponding IDL specification of the output interface is shown in Figure 4.3.

A nonterminal in the abstract syntax corresponds to a class in the IDL specification with the productions as its sub-classes. The names of the attributes in the IDL specification are the same as the attribute names in the specification of the editor. In the abstract syntax specification 'exp' is a nonterminal with a number of productions.

In the IDL specification, 'exp' is a class containing a number of nodes. Each of these nodes corresponds to a production of 'exp' and is named from it. The nonterminal 'exp' has an attribute 'v'. The class 'exp' has the corresponding attribute 'v'. The non-terminals on the right hand side of a production in the abstract syntax are considered as attributes of the non-terminal on the left hand side of a production. The names of these attributes are derived from the name of the production in the abstract syntax. The production 'Sum' is made of two occurrences of the nonterminal 'exp'. The node 'Sum' has two attributes 'Sum1' and 'Sum2' each of type 'exp'. Other schemes for naming the attributes may be used e.g., using the name of the non-terminal on the left hand side of the production. If a production does not have any non-terminals on the right hand side then it has no attributes. The production 'Null' has no non-terminals in it. The node 'Null' has no attributes. Even though the IDL specification has been generated by hand, it can be generated from the specification of the editor mechanically. The complete specification of the syntax-directed editor and the IDL specification of its output is provided in Appendix B.1.

Building a new output interface was fairly simple as predicted by the methodology. The output interface is a subroutine that outputs the IDL structure of the portion of the edited text that has been selected. This structure in ASCII ERL is output to the file 'foo'. This subroutine replaces the function 'dump-on' of the editor that printed the values of the attributes of the current selection. The editor can be modified to make this subroutine a new function. The code for the output interface, provided in Appendix B.2 was 150 lines long and took about 3 days to develop. It was simple because the traversal of the data structure was simple. Since the data structure had simple access macros, the traversal was not complicated. The number of attributes in a production and names of the different attributes were available in the internal data structure of the tool itself, therefore they were not hard-coded into the output interface. Because the data structures generated by the Synthesizer Generator are similar for different editors, this new output interface can be used by other editors as well.

4.2 Transformation of the input interface

This section considers the transformation of the input interface of the editor. The editor must be able to accept the IDL instance that it produces as output. The problem is to transform the input interface of the editor so that it accepts as input an instance of the IDL specification mentioned in Appendix B.1. This instance could have been produced by the new output interface constructed in Section 4.1. For this problem, the value along the Input or Output dimension is input. Along the Constraints of Implementation dimension, the external data format of the tool is known and the source code is available for modification.

Consider the IDL specification of the input data given in Figure 4.3. Some of the attributes of a node correspond to nonterminals in the abstract syntax (e.g. 'Sum1', 'Sum2' etc.,). A *syntactic attribute* is an attribute of a node that corresponds to a nonterminal in the abstract syntax. Some of the attributes correspond to attributes

of nonterminals in the abstract syntax (e.g. 'v'). A *semantic attribute* is an attribute that corresponds to an attribute of a nonterminal or to an attribute of a production in the specification of the editor.

Consider the transformation between the attribute tree represented as an IDL instance and the input text of the editor. Subtrees contained in the IDL instance correspond to contiguous portions of text. Independent subtrees correspond to independent portions of text. Hence, the 'Complexity of Computation' is 'simple'.

Consider the transformation between the attribute tree represented as an IDL instance and the structure input of the editor. The structure input to the editor is essentially a list of the operators in the abstract syntax tree listed in preorder. The attributed tree is a representation of the abstract syntax tree with attributes associated with some nodes in the tree. Therefore, the operators for two independent subtrees correspond to independent portions of the list. Therefore, the 'Complexity of Computation' is 'simple'.

The part of the internal data structure that contains the attributed syntax tree can be computed from the IDL instance. The IDL instance contains all the attributes required to build the attributed syntax tree. However, the attributed syntax tree contained in the internal data structure contains a set of flags associated with each node in the tree. Some of these flags are used to evaluate the attributes efficiently. These attributes should be computed fairly easily. Then, the 'Complexity of Computation' is 'simple'.

Summarizing, this problem maps to three points in the problem space each of which have along the 'Complexity of Computation' dimension, the value 'simple', two of the three points have along the 'Data' dimension the value 'external', and the other has along the 'Data' dimension, the value 'internal'. From the mapping given in Chapter 3, Approaches 1, 2, 5, 6 and 7 are applicable. The metrics on the transformation of the interface of the tool using the different approaches must be considered to choose the most appropriate approach from these approaches

Approach 1 may be applied to transform the input IDL instance to the textual or the structure input of the editor. The structure input of the editor contains the operators in the abstract syntax tree listed in preorder. The input IDL instance contains the attributed syntax tree in preorder. This IDL instance can be parsed and whenever a syntactic attribute is encountered, the operator corresponding to it is output. At the end of processing, the set of operators (along with their name and arity) is prefixed to the output. If the attributes of the nodes do not appear in order, then the list of operators must be stored and output at the end of processing.

The textual input of the tool can be generated by building the textual representations of the syntactic attributes in a bottom up fashion. The textual representation of a syntactic attribute which is a node is calculated from the textual representation of all syntactic attributes of the node using the appropriate unparsing rule. If the syntactic attribute were a node without any attributes or one of the basic types, then its textual representation is straightforward. The input IDL instance can be parsed and the representations can be computed bottom up. The solution is not affected by the order of the attributes in the IDL instance. Though both methods are fairly

simple, it is easier to generate the structure input to the editor.

Approach 2 may be applied to transform the input IDL instance to the textual or the structure input of the tool. In both cases the in memory IDL structure is created by reading in the IDL instance using routines provided by IDLC. To produce the structure input of the editor, the syntactic attributes are traversed in preorder and they are output. To produce the textual representation, the tree is traversed bottom up and the textual representation is built as specified in the previous section. Both these solutions are fairly straightforward, but the traversal of the in-memory IDL instance is complicated, since a separate routine is required for each node type in the IDL specification. This is fairly complicated even for a small specification. For an editor generated from a more complex language, it would be more complicated. These transformations are more complex than those that use Approach 1.

Approaches 5, 6 and 7 may be applied to build the attributed syntax tree portion of the internal data structure from the IDL instance. However associated with every node in the abstract syntax tree is a set of flags. Some of these flags are used to evaluate the attributes more efficiently. Since there is no documentation available about the internal data structure, it is not clear how these flags are computed. Further, errors in the evaluation of these flags are harder to detect since the editor is not well equipped to handle inconsistencies in its internal data structure. Therefore, transformation using these approaches is complicated. Approaches 5 and 6 may be applied to build the attributed syntax tree in preorder. Approach 7 may be used to build an IDL structure that takes the place of the attributed syntax tree contained in the internal data structure of the editor. The other parts of the internal data structure will not be changed, for e.g., arrays that contains the properties of the non-terminals. The code and the macros that access the attribute syntax tree will have to be modified appropriately.

The metrics on the different transformed editors (i.e. transformed using different approaches) are discussed in Chapter 3. Judging by the difficulty of the transformation it seems best to generate the structure input of the editor using Approach 1.

A filter was built that parsed the IDL instance (using YACC, see Figure 4.4) and output the list of operators in preorder into a file. The function 'check_syntactic_attribute' checks to see if the attribute whose name has been encountered is a syntactic or a semantic attribute. If it is a syntactic attribute, a flag is set. Subsequently when the operator associated with the attribute is encountered, the operator is output using the 'output_op' function. If the attribute value is a constant, then it is output using the function 'output_basic_type' function. The filter also stores the set of operators encountered and this set was output to a separate file. The latter file was prefixed to the former and this was provided as input to the editor. This filter was fairly easy to build. The code, presented in Appendix B.3 is about 330 lines long and it took about 1½ days to develop.

```

%start      value
%%
value       : TCONST { if syntactic_attr($1) output_basic_type($1);}
             | operator
             | operator TLBRAC attributes TRBRAC
             ;
operator    : TID { if syntactic_attr($1) output_op($1)};
attributes  : attributes TSEMICOLON attribute
             | attribute ;
attribute   : name value ;
name        | TID { check_syntactic_attribute($1)} ;
%%

```

Figure 4.4: Parser used to build filter

4.3 Conclusions

In this chapter the methodology given in Chapter 3 was applied to transform an input interface and an output interface. The methodology recommended a number of different approaches. Each approach provided a method of transforming the interface that had associated with it a number of metrics. The approach that provided the easiest transformation was adopted for implementation. The two interfaces were transformed and the results were as predicted. For instance, the metrics on the transformation predicted methods using Approaches 1, 2, 3 and 4 would be developed much faster. The output interface was transformed using Approach 8 and the input interface was transformed using Approach 1. Even though much less code was developed to transform the output interface, the input interface was transformed in half as much time as it took to transform the output interface.

The editor generated by the Synthesizer Generator consists of a set of core routines that perform all the basic functions of the editor. Those functions that are particular to an editor (e.g. the unparsing rules) are encoded in a table and the table is interpreted whenever those functions are required. The new output interface that has been built is capable of producing the attributed syntax tree in IDL format from any editor generated by the Synthesizer Generator. Similarly, the filter generated to transform an output interface is capable of producing a structure input from the attributed syntax tree in IDL format for any editor generated by the Synthesizer Generator.

Chapter 5

Transforming XDR

This chapter considers the application of the methodology to the External Data Representation (XDR) protocol [XDR 1986]. XDR will be examined to see how it fits the model of a tool. The methodology will then be applied to the transformation of the input interface of XDR. Next, the methodology will be applied to the transformation of the output interface of XDR. The chapter will conclude from the experience gained from these two transformations.

The International Standards Organization (ISO) has developed a seven-layered model for Open Systems Interconnection (OSI) [Tanenbaum 1981A]. This model describes the level of interaction in data communication between two computer systems. The sixth layer, the *presentation layer*, provides functions such as data encryption, data compression and other transformations on data that would be communicated between the systems. XDR is a presentation layer protocol, using which two heterogeneous machines can exchange in-memory data structures.

XDR is a protocol developed by Sun Microsystems Inc (see Figure 5.1). XDR converts between the machine dependent formats in which the data may be represented. Using XDR, two heterogeneous machines can exchange data structures specified in the 'C' programming language. XDR can be used to exchange data structures constructed from the basic 'C' types (`int`, `char`, etc.) using constructors for arrays, records and unions. Pointers can also be exchanged, but in a limited way. Data structures containing multiple pointers to the same location cannot be exchanged directly using XDR.

Even though XDR is very different from the syntax-directed editor considered in Chapter 4, it can be considered as a tool. The editor read in an expression as its input, and output an expression after processing it using commands from the user. XDR takes as input an in-memory 'C' data structure on one machine and provides as output the same data structure on a different machine.

The editor performs some transformations on its input to compute its output. XDR on the other hand moves the data structure from one machine to another. There were two objectives in the transformation of the editor, the first, to communicate with more tools (i.e. use a standard format) and the second, to communicate more information. XDR will be transformed so that it can communicate IDL instances between machines.

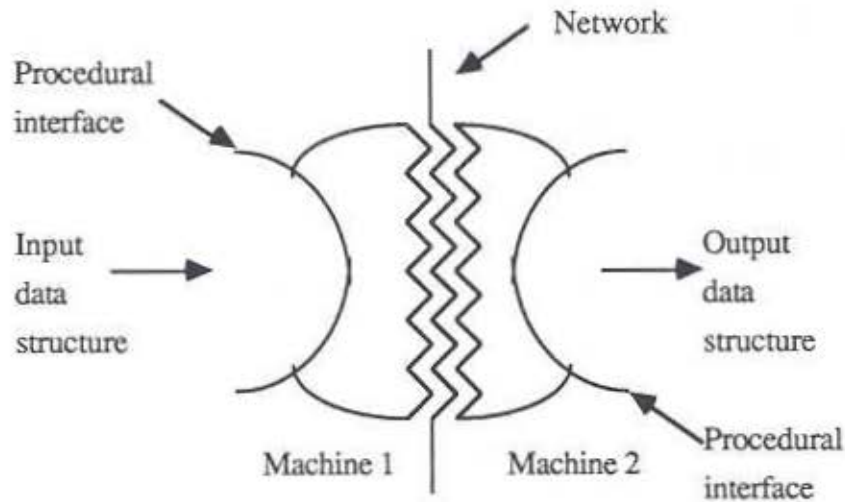


Figure 5.1: XDR

Unlike the editor that reads and writes data into files, XDR has a procedural interface, i.e., a data structure is communicated by XDR by calling a subroutine. The same subroutine can be used to send and receive the data structure. This routine calls the routines provided by XDR to communicate the basic types and some constructors. At one end, the 'C' data structure is converted by the subroutine into a series of bytes in a machine-independent format. These bytes are then transmitted to the other machine. At the other end, these bytes are converted into the data structure. The direction of data communication determines the type of conversion.

This procedural interface will be considered as the input and output interface of XDR. The format of the data structure associated with this interface is the format of the interface. The data structure created by routines generated by IDLC could have multiple pointers to the same location. These data structures cannot be directly communicated using XDR. Therefore the data structure declarations produced by IDLC cannot be the format of the interface.

The methodology requires that input and output data structures of the transformed tool be specified in IDL. For the purpose of transforming XDR a simple IDL specification was chosen. This specification is provided in Figure 5.2. This specification contains the basic types of IDL as well as the constructors.

The methodology considers the internal data structure of the tool to be transformed. It is not obvious what should be considered as the internal data structure of XDR. The only data structures maintained by XDR are buffers that are used to buffer the input and output data. However, these data structures store only a part of the input data. The representation of the input data structure as a sequence of bytes that are transmitted across the network can be considered as the internal data

```

Structure test Root Aval Is
  Aval => first: Cval,
         second: Dval,
         third: Seq Of Integer;
  Bval ::= Cval | Dval ;
  Bval => Enode: Eval;
  Cval => name: String;
  Dval => value: String;
  Eval => number: Integer,
        name: String,
        value: Rational,
        flag: Boolean;
End

```

Figure 5.2: IDL specification of the interfaces of the transformed tool

structure of XDR.

5.1 Transformation of the Input Interface

The input interface of XDR is a procedural interface that accepts a 'C' data structure and transmits it over the network to another machine. The data structure input to XDR is specified using IDL in Figure 5.2.

To apply the methodology, the characteristics of the problem along the different dimensions of the problem space have to be determined. The problem space has four dimensions: Input or Output, Data, Complexity of Computation and Constraints on the Implementation. The input interface is being transformed, therefore the value along the 'Input or Output' dimension is 'Input'. Along the 'Constraints on the Implementation' dimension, the format of the data input and output by the tool are available. The problem is multi-valued along the 'Data' dimension.

Consider the value 'external' along the 'Data' dimension. The 'C' data structure communicated by XDR cannot have multiple pointers to the same memory location, i.e. nodes may not be shared. In the IDL data structure, the node of type 'Eval' may be shared by the two nodes of the class 'Bval' (see Figure 5.3). Since this data structure may have to be communicated, the sharing of the node has to be encoded in the data structure communicated by XDR. This may be an attribute in the data structure communicated by XDR. The computation of this attribute requires the examination of the whole data structure. The 'Complexity of Computation' is 'Complex'.

Consider the value 'internal' along the 'Data' dimension, i.e. computation of the internal data structure from the IDL instance. The internal data structure is a representation of interface data structure. The computation of the internal data


```

-- structure test
Aval
  [ first Cval
    [ name "Sundar" ;
      Enode L10: Eval
        [ number 5 ;
          name "Sundar" ;
          value 5.0 ;
          flag 0 ;
        ]
      ];
    second Dval
      [ value "Sundar" ;
        Enode L10^ ];
    third < 1 2 >
  ]
#

```

Figure 5.3: An instance of the specification in Figure 5.2 in IDL format

structure requires the encoding of the sharing of nodes. This requires examination of the entire data structure. Therefore the 'Complexity of Computation' along the data dimension is 'Complex'.

Summarizing, this problem resolves to two points in the problem space. One of them has value 'external' and another the value 'internal' along the 'Data' dimension. Both points have, along the 'Input or Output' dimension, the value 'input' and, along the 'Complexity of Computation' dimension, the value 'Complex'. From the mapping given in Chapter 3, Approach 2 is applicable. Approaches 6 and 7 are not applicable since the source code is not available. The XDR protocol specifies the encoding of a 'C' data structure in a machine independent format to be communicated to another machine. Using this specification the IDL data structure can be converted to a format to be communicated to another machine. This would amount to a re-implementation of XDR, which is not considered as an approach by the methodology.

In Approach 2, the in-memory IDL structure is created by reading the input IDL instance using routines provided by IDLC. This structure is traversed and the input data of the tool is computed. In the case of XDR, this structure is traversed and the data structure that is required by the input interface of XDR is computed. Since XDR has a procedural interface, this data structure has to be computed by a subroutine. This application of Approach 2 is similar to Approach 6 in which the internal data structure of the tool is computed using a subroutine. The difficulty of the transformation depends on the difficulty of computing the data structure required by XDR.

The data structure to be supplied to XDR needs to be specified before the difficulty of the transformation can be evaluated. The data structure supplied to XDR will be computed by traversing the IDL data structure. The data structure can be computed easily if it is similar to the IDL data structure. This data structure has to encode the sharing of nodes. A data structure was chosen that was very similar to the IDL data structure. This data structure shown in Figure 5.4 encodes sharing of nodes of type 'Eval' using character strings as labels. Further, null pointers in the structure 'intseq' are implemented using discriminated unions, since XDR cannot communicate null pointers directly. The transformation is straightforward and fairly simple. However, for a more complex structure, the transformation would be complicated, given the nature of data declarations produced by IDLC.

The IDL structure is read into memory using routines provided by IDLC. The data structure required by the input interface can be computed in two phases. In the first phase the IDL structure is traversed and nodes of type 'Eval' are marked as touched. If a node marked as touched is encountered, it is marked as shared. In the second phase, the data structure is created from the IDL data structure. If a node marked shared is encountered, then it is labeled. Subsequent references to the node are encoded in the data structure as labels. The data structure is then communicated using XDR. The program provided in Appendix C.3 is about 120 lines long and took about a day to develop.

5.2 Transformation of the Output Interface

The data structure communicated across the input and output interfaces of XDR is the same. The output interface of XDR is a procedure that when called produces an instance of the data structure specified in Figure 5.4. This data structure needs to be converted into an equivalent instance of the IDL specification in Figure 5.2.

To apply the methodology, the characteristics of the problem along the dimensions of the problem space have to be determined. The problem space has four dimensions: Input or Output, Data, Complexity of Computation and Constraints on the Implementation. The output interface of XDR is being transformed, therefore the value along the 'Input or Output' dimension is 'Output'. Along the 'Constraints on the Implementation' dimension, the format of the data input and output by the tool are available. The problem is multi-valued along the 'Data' dimension.

Nodes of type 'Eval' may be shared in the IDL instance. This is encoded in the data structure using labels and label references. This is similar to the IDL format for representing nodes that are shared. There is a one to one correlation between the fields of the data structure and the IDL format. Therefore, for the value 'external' along the 'Data' dimension, the value along the 'Complexity of Computation' dimension is 'simple'. Recall that the value along this dimension in the transformation of the input interface was 'Complex'.

The internal data structure is a representation of the data structure of the output interface. The internal data structure contains shared nodes encoded using labels and label references. This is similar to the representation of the data structure in the

```

struct hE {
    enum {ISLABEL=1,ISNODE=2} Etype;
    union {
        int label_no;
        struct hEvalue {
            enum {NOLABEL=1,LABELDEF=2} Etype;
            int label_no;
            struct {
                int number;
                char *name;
                float value;
                int flag;
            } value;
        } value;
    } value;
};

struct hC {
    char *name;
    struct hE Enode;
};

struct hD {
    char *value;
    struct hE Enode;
};

struct intseq {
    int val;
    enum {NULLELEM=1,VALELEM=2} nodetype;
    struct intseq * next;
};

struct hA {
    struct hC first;
    struct hD second;
    struct intseq third;
};

```

Figure 5.4: Data structure of the interface specified in 'C'

```

struct hE {
    enum {ISLABEL=1,ISNODE=2} Etype;
    union {
        int label_no;
        struct hEvalue {
            enum {NOLABEL=1,LABELDEF=2} Etype;
            int label_no;
            struct {
                int number;
                char *name;
                float value;
                int flag;
            } value;
        } value;
    } value;
};

struct hC {
    char *name;
    struct hE Enode;
};

struct hD {
    char *value;
    struct hE Enode;
};

struct intseq {
    int val;
    enum {NULLELEM=1,VALELEM=2} nodetype;
    struct intseq * next;
};

struct hA {
    struct hC first;
    struct hD second;
    struct intseq third;
};

```

Figure 5.4: Data structure of the interface specified in 'C'

IDL format. Therefore, for the value ‘internal’ along the ‘Data’ dimension, the value along the ‘Complexity of Computation’ dimension is ‘simple’.

From the mapping given in Chapter 3, Approaches 3 and 4 are applicable. Approaches 8, 9 and 10 are not applicable since the source code is not available. As in the transformation of the input interface, the re-implementation of XDR is not considered.

Approach 4 can be applied to build the IDL data structure internally. The IDL data structure represents sharing of nodes by using multiple pointers to the same location. The interface data structure encodes the sharing of nodes using labels and label references. A table of labels and their associated pointers has to be constructed. Since labels may be encountered in the traversal of the internal data structure before they are defined, the table may also need to contain references to all references to the label. The interface data structure is traversed depth-first and the IDL data structure is built top-down. If a field in the interface data structure is not a reference to a label, then the node corresponding to it is built. If the field is shared, it may define a label that would be referenced later. Then the table is updated with the label and a reference to the newly built node. If the field is a reference to a label, then its value is obtained by looking up the table. References to labels that are not yet defined will have to be handled as well. This approach is straightforward, but complicated. For a more complex structure it would be more complicated.

Approach 3 can be applied by traversing the interface data structure and outputting the attributes of the different nodes in the IDL format. The interface data structure has been created by a call to an XDR routine. If a node of type ‘Eval’ is shared, then it is encoded in the data structure using labels and label references. This is similar to the IDL format for encoding nodes that are shared. A subroutine that outputs the interface data structure in the IDL format was written. Since the interface data structure did not contain the names of the fields, they were coded into the subroutine. This approach is straightforward and simple. For a more complex interface data structure, this approach would be more complicated. This approach was implemented. The code, provided in Appendix C.4, is 66 lines long and took about $\frac{1}{2}$ a day to develop.

5.3 Conclusions

This chapter discussed the application of the methodology to XDR. Since XDR is very different from a conventional tool like the syntax-directed editor, the transformation was unconventional. Conventionally, this transformation would have been viewed as the problem of representing an IDL instance in XDR. In that case, the issues would have been a compact and efficient representation using XDR. However the methodology required the specification of the IDL structure and the specification of XDR’s interface were specified a priori. Therefore this methodology cannot be used to directly measure the efficiency of the representation in XDR.

Another issue that would have been considered would be to automate this transformation. That is, given the IDL specification, the specification of the XDR interface

and the routines to communicate the IDL instance be done mechanically.

In this transformation the XDR format of the interface was specified. The format of the XDR interface could have been specified as a stream of characters. In that case, the instance in ASCII ERL can be communicated directly. However, this would have defeated the purpose of XDR which is meant to communicate data structures in machine dependent format between heterogeneous machines.

Chapter 6

Conclusions

This research provides a methodology to transform the the input and output interfaces of an existing tool to communicate instances in the IDL format. The characteristics of the problem are used to choose the approaches appropriate to the problem. The approaches can be evaluated using the metrics associated with them. The methodology was applied to two problems, transforming the input and output interfaces of a syntax directed editor and transforming XDR to communicate IDL instances. The results of these applications provide testimony to the validity of the methodology.

This research provides a systematic way to transform tools to communicate IDL structures and to evaluate the cost of such a transformation. The transformed tool may be able to communicate with many other tools. The transformed tool can also be tested and debugged using the IDL toolkit.

6.1 Future Work

As mentioned in Chapter 3, this research did not consider how the IDL specification of the interfaces may be derived from the tool. The IDL specification of the interfaces of the syntax-directed editor could have been automatically generated from the specification of the editor. The attributes of a node in the IDL specification are the syntactic and semantic attributes of the corresponding production in the abstract syntax. The IDL specification cannot be derived automatically for all tools that are specified formally. It would illuminating to consider the aspects of the specification that make such a derivation possible.

After an IDL specification has been derived, it needs to be evaluated to determine if it contains all the attributes that would be required by other tools that communicate the IDL instance. This is dependent on the environment in which that transformed tool is to function.

In the transformation of XDR, it was noted that the transformation could be automated. This needs to be examined. In general, automating the transformation would be be very difficult, if not impossible.

The methodology considers the transformation of just one interface at a time. There may be advantages in considering the transformation of the tool as a whole.

The new tool may be more efficient and also more easily maintainable. In some cases, even the transformation may be easier. This needs to be investigated further.

The methodology assumes that the tool reads in all the input data into its internal data structure, performs computations on the data, and then produces the output data from its internal data structure. Some tools may process the input incrementally. Some tools may encode the information about the input data structure in their program state (e.g. recursive descent parsers). The methodology may not be able to provide a practical solution for such problems. Specialized techniques may be considered.

As stated in Chapter 1, the transformation of IDLC to accept IDL instances in tool dependent formats was not considered. In some cases, it may be cost effective to build a filter that converts between the tool dependent format of the IDL instance and the IDL format.

Chapter 7

Bibliography

- [Biyani 1987] Biyani, V. *An Efficient Runtime System for IDL*. Master's Thesis, University of North Carolina at Chapel Hill, Oct. 1987.
- [Chambers, et al. 1984] Chambers, F., D. Duce and J. Gillian. *Distributed Computing*. London: Academic Press, 1984.
- [Hopcroft & Ullman 1979] Hopcroft, J.E. and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.
- [Kernighan & Pike 1984] Kernighan, B.W. and R. Pike. *The UNIX Programming Environment*. Englewood Cliffs, NJ 07632: Prentice-Hall, 1984.
- [Kernighan & Ritchie 1988] Kernighan, B.W. and D.M. Ritchie. *The C Programming Language*. Vol. second edition of Prentice Hall Software Series. Englewood Cliffs, NJ: Prentice Hall, 1988.
- [Nestor et al. 1982] Nestor, J.R., W.A. Wulf and D.A. Lamb. *IDL - Interface Description Language - Formal Description - Draft Revision 2.0*. Internal Document. Computer Science Department, Carnegie-Mellon University. June 1982.
- [Reps & Teitelbaum 1984] Reps, T. and T. Teitelbaum. *The Synthesizer Generator*, in *Proceedings of the Symposium on Practical Software Development Environments*, Pittsburgh, PA: Apr. 1984, pp. 42-48.
- [Snodgrass 1988] Snodgrass, R. *The Interface Description Language: Definition and Use (forthcoming)*. Rockville, MD: Computer Science Press, 1988.
- [Tanenbaum 1981B] Tanenbaum, A. S. *Network Protocols*. *ACM Computing Surveys*, 13, No. 4, Dec. 1981, pp. 453-489.

- [Tanenbaum 1981A] Tanenbaum, A.S. *Computer Networks*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.
- [Wirth 1971] Wirth, N. *The Programming Language Pascal*. *Acta Informat.*, 1 (1971), pp. 35-63.
- [XDR 1986] *External Data Representation Protocol Specification*. Sun Microsystems Inc., Mountainview, CA, 1986.

Appendix A

Glossary

IDL format The standard format (ASCII ERL) in which the IDL structure is read or written to external storage.

IDL Interface Description Language. A language developed to communicate data structures between programs.

Synthesizer Generator A program that can be used to generate syntax-directed editor. For more details see Chapter 4.

XDR A protocol developed by Sun Microsystems Inc., to communicate 'C' data structures between heterogenous machines.

abstract syntax tree The derivation of a sentence in the abstract syntax represented as a tree.

abstract syntax A set of productions that specifies the syntax of a language without the use of terminal symbols.

attribute A named value whose domain is specified by its type. In IDL, an attribute is conceptually similar to a field of a Pascal record. In the specification of a syntax-directed editor, attributes are attached to either non-terminals or productions.

attributed syntax tree The abstract syntax tree along with the semantic attributes associated with the non-terminals and operators in the tree.

attribution rule In the specification for the Synthesizer Generator, a rule used to compute the value of an attribute.

complex computation A computation that is not simple.

external IDL instance The IDL data structure represented in the IDL format.

- filter** A program that converts the input or output data between the tool dependent format and IDL format.
- interface** The format of data and the set of routines that read and write the data.
- internal IDL instance** The IDL data structure represented in memory.
- node** A data type in IDL that is a named collection of attributes. This is similar to a Pascal record.
- operator** In the specification of a syntax-directed editor, the name of a production in the abstract syntax.
- parsing declaration** The rule used to generate a portion of the abstract syntax tree from the input text. A set of parsing declarations specify how the abstract syntax tree may be computed from the input text.
- pipe** A mechanism provided by the UNIX operating system to communicate the output of one program as the input of another program without using intermediate files.
- presentation layer** The sixth layer in the ISO's OSI model that provides functions such as data encryption, data compression and other transformations on data.
- root** A distinguished node in the IDL structure from which all other nodes may be accessed.
- semantic attribute** When the attributed syntax tree is represented as an IDL structure, the attributes in the IDL structure that correspond to attributes of the non-terminals and operators.
- simple computation** A computation of the IDL instance from the internal data structure where there is correspondance between attributes in the IDL specification and portions of the internal data structure. For a more complete definition see Section 3.1.
- structure format** The representation of the abstract syntax tree, with the list of operators with their name and arity, followed by the operators in the abstract syntax tree listed in preorder.
- syntactic attribute** When the attributed syntax tree is represented as an IDL structure, the attributes in the IDL structure that correspond to non-terminals in the abstract syntax.
- syntax-directed editor** A text editor where the text edited is constrained to be a program in a language. This may be generated using the Synthesizer Generator.
- textual format** The textual representation of the abstract syntax tree using the unparsing rules.

tool dependent format The format of the input and output data. This format is usually dependent on the tool.

tool A program that has an input and an output.

unparsing rule A rule that used to generate the textual representation of a production in the abstract syntax tree.

Appendix B

Code for Syntax-directed Editor

Chapter 4 describes the transformation of a syntax directed editor. This appendix contains the IDL specification of the new input and output interfaces and the code needed for the transformation.

B.1 The Specification

This section contains a part of the specification of the syntax directed editor, with the abstract syntax and the attributes. The complete IDL specification resembles the editor's specification rather closely. The specification of the editor provided here is not the same as that distributed with the Synthesizer Generator.

The abstract syntax specification of the editor with attributes

```

/* Abstract syntax */
root calc;
list calc;
calc : CalcPair(exp calc)
      | CalcNil()
      ;

exp : Null()
     | Sum, Diff, Prod(exp exp)
     | Quot(exp exp) { local STR error;}
     | Const(INT)
     | Let(symb exp exp)
     | Use_id(ID) { local BINDING b; local STR error;}
     ;

symb : DefBot()
      | Def(ID)
      ;

/* Semantic attributes of the non-terminals */
exp { synthesized INT v;};
exp { inherited ENV env; };
symb { synthesized ID id; };

/* Type definition for environments */
list ENV;
ENV : NullEnv()
     | EnvConcat(BINDING ENV)
     ;
BINDING : Binding(ID INT);

```

The complete IDL specification

```

Structure exp_tree Root calc Is
calc ::= CalcPair | CalcNil ;
CalcPair => CalcPair1: exp,
           CalcPair2: calc ;
CalcNil => ;
exp ::= Null | Sum | Diff | Prod | Quot | Const | Let | Use_id ;
Null => ;
Sum => Sum1: exp,
      Sum2: exp ;
Diff => Diff1: exp,
      Diff2: exp ;
Prod => Prod1: exp,
      Prod2: exp ;
Quot => Quot1: exp,
      Quot2: exp,
      error: String ;
Const => Const1: Integer ;
Let => Let1: symb,
     Let2: exp,
     Let3: exp ;
Use_id => Use_id1: String,
        b: BINDING,
        error: String ;
symb ::= DefBot | Def ;
DefBot => ;
Def => Def1: String ;

-- Semantic attributes.
exp => v: Integer ;
exp => env: ENV ;
symb => id: String ;

-- Symbol table
ENV ::= NullEnv | EnvConcat ;
NullEnv => ;
EnvConcat => EnvConcat1: BINDING,
            EnvConcat2: ENV ;
BINDING ::= Binding ;
Binding => Binding1: String,
          Binding2: Integer ;
End

```


B.2 Transformation of the Output Interface

The output interface was transformed using Approach 8. The subroutine that was coded 'MET_print_ERL' is presented in this section. The function 'dump-on' of the editor was modified so that the subroutine 'dump_atree' calls the subroutine 'MET_print_ERL' with the current selection.

File : test.c

page 1

```
#include "lang.h"
#include "structures.h"
#include "grammar.h"
#include "attr.h"
#include "types.h"
#include "atree.h"
#include "selection.h"
#include "buffers.h"
#include "dequeue.h"
#include "output.h"
#include "display_map.h"
#include "viewport.h"
#include "browser_exp.h"
#include "hash_table.h"
#include "edit_buf.h"
#include <stdio.h>
#include <strings.h>

PROCEDURE MET_print_ERL(tree,filename)
    register ATREE tree;
    char *filename;
    /* Open the file and gets things ready for printing */
{
    register PROD_INSTANCE start_p;
    char t[100];
    int i;
    FILE *fp, *fopen();

    fp=fopen(filename,"w");
    if (fp==NULL) {
        fprintf(stderr, "Can't open file %s for writing\n", filename);
        return;}

    start_p = selection_apex(selection(tree));
    MET_print_parse_tree(fp,start_p,0);
    fclose(fp);
}
```

File : test.c

page 2

```

PROCEDURE MET_print_parse_tree(fp,p,is_attr)
  FILE *fp;
  register PROD_INSTANCE p;
  int is_attr; /* 1= is an attribute, 0 is a parse-tree */

  /* Given a parse tree print all the sons and all the attributes */
  /* associated with the tree. */

{
  char t[100], *MET_comp_son_name();

  if (atom(production(p))) MET_print_atom(fp,p);
  else
  { int i;
    char *s;
    s=op_name(production(p));
    sprintf(t, "%s\n",s);
    MET_print_string(fp,t);
    if ((!is_attr && (no_attrs(lhs_occ(production(p)))>0)) ||
        (no_sons(production(p))>0))
        MET_print_string(fp,"[");
    if (!is_attr && (no_attrs(lhs_occ(production(p)))>0))
    { MET_print_attr_tree(fp, p);
      if (no_sons(production(p))>0) MET_print_string(fp,";\n");
    }
    for (i=1;i<=rightmost_son(production(p));i++)
    { PROD_INSTANCE q;
      sprintf(t,"%s \n",MET_comp_son_name(s,i));
      MET_print_string(fp,t);
      q=son(p,i);
      MET_print_parse_tree(fp,q,is_attr);
      if (i!=rightmost_son(production(p))) MET_print_string(fp,";\n");
    }
    if ((!is_attr && (no_attrs(lhs_occ(production(p)))>0)) ||
        (no_sons(production(p))>0))
        MET_print_string(fp,"]");
  }
}

```

File : test.c

page 3

```

static PROCEDURE MET_print_attr_tree(fp, p)
    FILE *fp;
    register PROD_INSTANCE p;

    /* Given a parse tree p print all attributes of p, both local to the
       production as well as attributes of the left hand side */

{
    char t[100];
    int k;
    ATTR current_attr;

    current_attr = fld_index(lhs(production(p)));
    for (k = 0; k < no_attrs(lhs_occ(production(p)));) {
        MET_print_string(fp, "\n");
        if (k == no_fields(sym_index(lhs_occ(production(p))))
            current_attr = local_fld(lhs_occ(production(p)));

        sprintf(t, "%s ", fld_id(current_attr));
        MET_print_string(fp, t);
        current_attr++;
        MET_print_attr(fp, attr_instance(p, 0, k));
        if (++k < no_attrs(lhs_occ(production(p)))) MET_print_string(fp, "; \n");
    }
}

/*
 * MET_print_attr
 *
 * Print value of ATTR_INSTANCE b to FILE fp
 */
static PROCEDURE MET_print_attr(fp, b)
    FILE *fp;
    ATTR_INSTANCE b;

{
    register PROD_INSTANCE attr_v;
    attr_v = demand_value(b); /* ensure that value of b is available */
    if (attr_v == (PROD_INSTANCE) 0)
        MET_print_string(fp, "-- Null Value\n");
    else
        MET_print_parse_tree(fp, attr_v, 1);
}

```

File : test.c

page 4

```

/* Given an atomic PROD_INSTANCE p, print p to file fp */

MET_print_atom(fp, p)
    FILE *fp;
    PROD_INSTANCE p;
{ char t[50];
  char *str0_to_str();
  switch ( atomic_type(production(p)) ) {
  case 0: sprintf(t, " %d\n", IntValue(p)); break;
  case 1: sprintf(t, " %d\n", RealValue(p)); break;
  case 2: sprintf(t, " %d\n", DrealValue(p)); break;
  case 3: sprintf(t, " \'%c\'\n", CharValue(p)); break;
  case 4: sprintf(t, "\"%s\"\n", ((BoolValue(p))?"TRUE":"FALSE")); break;
  case 5: MET_print_string(fp, "");
    MET_print_string(fp, str0_to_str(son(p,1)));
    MET_print_string(fp, "");return;
  case 6: MET_print_string(fp, "-- Ptr not supported\n"); return;
  case 7: MET_print_string(fp, "-- Attr not supported\n"); return;
  case 8: MET_print_string(fp, "-- Hash table not supported\n"); return;
  }
  MET_print_string(fp,t);
}

char * MET_comp_son_name(s,i)
    char *s;
    int i;
    /* Compute the name of the ith non terminal in a production. */
{ static char t[50];
  if (*s=='\0') return(s);
  sprintf(t,"%s%d",s,i);
  return(t);
}

MET_print_string(fp,t)
    FILE *fp;
    char *t;
    /* Print the string to the output file */
{ fprintf(fp,"%s",t);}

```

B.3 Transformation of the Input Interface

The input interface was transformed using Approach 1. A lexical analyser (using LEX) and a parser (using YACC) were built to parse the input. The parser is called with 2 file names. At the end of execution the first file contains the name and arity of the different operators. The second file contains the operators listed in preorder.

File : test.c

page 4

```

/* Given an atomic PROD_INSTANCE p, print p to file fp */

MET_print_atom(fp, p)
    FILE *fp;
    PROD_INSTANCE p;
{ char t[50];
  char *str0_to_str();
  switch ( atomic_type(production(p)) ) {
  case 0: sprintf(t, " %d\n", IntValue(p)); break;
  case 1: sprintf(t, " %d\n", RealValue(p)); break;
  case 2: sprintf(t, " %d\n", DrealValue(p)); break;
  case 3: sprintf(t, " \'%c'\n", CharValue(p)); break;
  case 4: sprintf(t, "\'%s'\n", ((BoolValue(p))?"TRUE":"FALSE")); break;
  case 5: MET_print_string(fp, "");
          MET_print_string(fp, str0_to_str(son(p,1)));
          MET_print_string(fp, "");return;
  case 6: MET_print_string(fp, "-- Ptr not supported\n"); return;
  case 7: MET_print_string(fp, "-- Attr not supported\n"); return;
  case 8: MET_print_string(fp, "-- Hash table not supported\n"); return;
  }
  MET_print_string(fp,t);
}

char * MET_comp_son_name(s,i)
    char *s;
    int i;
    /* Compute the name of the ith non terminal in a production. */
{ static char t[50];
  if (*s=='\0') return(s);
  sprintf(t,"%s%d",s,i);
  return(t);
}

MET_print_string(fp,t)
    FILE *fp;
    char *t;
    /* Print the string to the output file */
{ fprintf(fp,"%s",t);}

```

B.3 Transformation of the Input Interface

The input interface was transformed using Approach 1. A lexical analyser (using LEX) and a parser (using YACC) were built to parse the input. The parser is called with 2 file names. At the end of execution the first file contains the name and arity of the different operators. The second file contains the operators listed in preorder.

File : main.c

page 1

```

#include <stdio.h>
extern FILE *fp1, *fp2;
main(argc,argv)
    int argc;
    char *argv[];
{ FILE *fopen();
  int token;
  if (argc!=3)
    { fprintf(stderr,"usage:%s file1 file2\n",argv[0]);exit(99);}
  fp1=fopen(argv[1],"w");
  if (fp1==NULL)
    { fprintf(stderr,"Can't open %s\n",argv[1]); exit(99);}
  fp2=fopen(argv[2],"w");
  if (fp2==NULL)
    { fprintf(stderr,"Can't open %s\n",argv[2]); exit(99);}
  yyparse();
  print_opsers();
}

yyerror(s)
    char *s;
{ printf("parser error %s\n",s);}

```

File : yacc.h

page 1

```

#include <stdio.h>
extern struct const_elem {
    int tag; /* 0 = integer, 1 = string, 2 = rational, 3 = boolean */
    union {
        int int_value;
        char * str_value;
        float real_value;
        int bool_value;
    } value;
    } const_array[500];

extern int no_of_consts,no_of_ids,no_of_opsers,lineno;
extern int const_id[4];

extern struct id_elem {
    char *name;
    int operator; /* number >=0 if operator */
    } id_array[500];

extern struct oper_elem {
    int id; /* pointer into id_array */
    int arity; /* number >=0 indicates proper arity */
    } oper_array[500];

```

```
/* maintain a stack of currently active operands */  
extern int curr_oper[500],curr_len[500],curr_arity[500],op_depth;  
extern FILE *fp1,*fp2;  
extern int is_attr,sig;
```

File : gram.y

page 1

```
%{
#include "yacc.h"
%}
%start value
%term CONST TID LBRAC RBRAC SEMICOLON
%%
value : CONST
      { output_const($1); }
    | TID
      { handle_0_op($1);}
    | TID { setup_n_op($1);} LBRAC attributes RBRAC
      { winddown_n_op();}
    ;
attributes : attributes SEMICOLON attribute
           | attribute
           ;
attribute : TID { start_son($1);} value
          {end_son($1);}
          ;
%%
```

File : gram.y

page 2

```

output_const(n)
    int n;
    /* output constants */
{ char t[1000];
  if (!is_attr) {
    fprintf(fp2,"%d\n",id_array[const_id[const_array[n].tag]].operator);
    switch(const_array[n].tag){
    case 0 : {
      sprintf(t,"%d",const_array[n].value.int_value);
      fprintf(fp2,"%d %s\n",strlen(t),t);
    }; break;
    case 1 : {
      char *s;
      int i;
      s=const_array[n].value.str_value;i=0;
      while (*s != '\0' )
        { int code;
          code = *s & 0x7f;
          if ( code == 0x7f || code == 0x5c || code < 0x21 )
            { char str[10];
              sprintf(str,"\\%3o ",code);t[i]='\0';
              strcat(t,str);i=i+5;
            }
          else t[i++]= *s;
          s++;
        }
      t[i]='\0';
      fprintf(fp2,"%d %s\n",strlen(s),t);
    }; break;
    case 2 : {
      sprintf(t,"%e\n",const_array[n].value.real_value);
      fprintf(fp2,"%d %s\n",strlen(t),t);
    }; break;
    case 3 : {
      if (const_array[n].value.bool_value)
        fprintf(fp2,"4 true\n");
      else fprintf(fp2,"5 false\n");
    }; break;
    default : fprintf(stderr,"What tag %d\n",const_array[n].tag);break;
    }
  }
}
}

```

File : gram.y

page 3

```

handle_0_op(n)
    int n;

    /* Handle an operator of arity 0 */

{ if (!is_attr) {
  if (id_array[n].operator >= 0 )
    /* previously defined as an operator */
    if (oper_array[id_array[n].operator].arity != 0)
      {
        fprintf (stderr,"operator %s has different arities\n",
                  id_array[n].name);
        fprintf (stderr,"Error occured about %d line\n",lineno);
        exit(99);
      }
    else ;
  else
    { id_array[n].operator=no_of_ops;
      oper_array[no_of_ops].id=n;
      oper_array[no_of_ops].arity=0;
      no_of_ops++;
    }
  fprintf(fp2,"%d\n",id_array[n].operator);
}
}

setup_n_op(n)
    int n;

    /* Set up to handle an operator of arity >0 */

{ if (!is_attr)
  { int opnum;
    if (id_array[n].operator>=0) opnum=id_array[n].operator;
    else {
      if (no_of_ops>=(sizeof(oper_array)/sizeof(oper_array[0])))
        { fprintf(stderr,"Too many operators\n"); exit(99);}
      id_array[n].operator=no_of_ops;
      oper_array[no_of_ops].id=n;
      oper_array[no_of_ops].arity= -1; /* arity not yet decided */
      opnum=no_of_ops;no_of_ops++;
    }
    op_depth++;
    curr_arity[op_depth]=0;
    curr_oper[op_depth]=opnum;
    curr_len[op_depth]=strlen(id_array[n].name);
    fprintf(fp2,"%d\n",opnum);
  }
}
}

```

File : gram.y

page 4

```

winddown_n_op()
    /* A production has been fully parsed */
{ if (!is_attr)
    { if (oper_array[curr_oper[op_depth]].arity>=0)
        if (oper_array[curr_oper[op_depth]].arity!=curr_arity[op_depth])
            { fprintf(stderr,"oper %s has different arity\n",
                id_array[oper_array[curr_oper[op_depth]].id].name);
              fprintf(stderr,"error is on line %d\n",lineno);
              exit(99);
            }
        oper_array[curr_oper[op_depth]].arity=curr_arity[op_depth];
        op_depth--;
    }
}

start_son(n)
    int n;
    /* A new attribute is being encountered */
{
    if (!(is_attr ||
        strcmp(id_array[n].name,
            id_array[oper_array[curr_oper[op_depth]].id].name,
            curr_len[op_depth])))
        curr_arity[op_depth]=curr_arity[op_depth]+1;
    else is_attr++;
}

end_son(n)
    int n;
    /* Attribute has been parsed, process appropriately. */
{ if (is_attr ||
    strcmp(id_array[n].name,
        id_array[oper_array[curr_oper[op_depth]].id].name,
        curr_len[op_depth]))
        is_attr--;
}

print_ops()
    /* print the list of operators with their name and arity */
{ int i;
    fprintf(fp1,"A#S#C#S#S#L#V#2\n");
    fprintf(fp1,"$operators \n");
    for(i=0;i<no_of_ops;i++)
        if (is_primitive(i))
            fprintf(fp1,"%s 0 1\n",id_array[oper_array[i].id].name);
        else
            fprintf(fp1,"%s %d 0\n",id_array[oper_array[i].id].name,
                oper_array[i].arity);
    fprintf(fp1,"$term \n");
}

```

File : gram.y

page 5

```
int is_primitive(n)
    int n;
    /* Check to see if the operator is one of the primitive operators */
{ int id,i;
  id=oper_array[n].id;
  for(i=0;i<4;i++)
    if (const_id[i]==id) return(1);
  return(0);
}
```

```
%{
#include <stdio.h>
#include "y.tab.h"
  struct const_elem {
    int tag; /* 0 = integer, 1 = string, 2 = rational, 3 = boolean */
    union {
      int int_value;
      char * str_value;
      float real_value;
      int bool_value;
    } value;
  } const_array[500];

  int no_of_consts=0,no_of_ids=0,no_of_ops=0,lineno=1;
  int const_id[4]= {-1,-1,-1,-1};

  struct id_elem {
    char *name;
    int operator; /* number >=0 if operator */
  } id_array[500];

  struct oper_elem {
    int id; /* pointer into id_array */
    int arity; /* number >=0 indicates proper arity */
  } oper_array[500];

  /* maintain a stack of currently active operands */
  int curr_oper[500],curr_len[500],curr_arity[500],op_depth= -1;
  FILE *fp2,*fp1;
  int is_attr=0;

  extern int yy1val;
%}
```



```

DIGITS      ([0-9]+)
DTDIGITS    (({DIGITS}"."|("."{DIGITS})|({DIGITS}"."{DIGITS}))
EXP         ([Ee][+-]?{DIGITS})
FLOAT1      ({DIGITS}{EXP})
FLOAT2      ({DTDIGITS}{EXP}?)
RATIONAL    (({FLOAT2})|({FLOAT1}))
INTEGER     ("+"|"-"?{DIGITS})
STRING      \"[-\\\"\\n]*\"
ID          [A-Za-z_][A-Za-z0-9_]*

%%
"["      return(LBRAC);
"]"      return(RBRAC);
{INTEGER} { yylval=add_const(yytext,0);
            return(CONST);
          }
{STRING}  { yylval=add_const(yytext,1);
            return(CONST);
          }
{RATIONAL} { yylval=add_const(yytext,2);
            return(CONST);
          }
{ID}     { if ((!strcmp(yytext,"TRUE")) || (!strcmp(yytext,"FALSE")))
            { yylval=add_const(yytext,3);
              return(CONST);
            }
            else { yylval=add_id(yytext);
                  return(TID);
                }
          }
";"      return(SEMICOLON);
[ \\t]   ;
.        fprintf (stderr,"error unrecognisable character '%c' on line %d\\n",
                *yytext,lineno);
"\\n"    lineno++;
%%

```

```

int add_const(yytext,tag)
    char *yytext;
    int tag; /* 0 = integer, 1 = string, 2 = rational, 3 = boolean */
    /* Add a constant to the list of constants */
{
    int value;
    char *malloc(),*s,*t;
    float real;
    struct const_elem *new_elem;

    /* first add the constant to the list of constants */

    if (no_of_consts >=(sizeof(const_array)/sizeof(const_array[0])))
        { fprintf(stderr,"Too many constants\n"); exit(99);}
    new_elem= &(amp;const_array[no_of_consts]);
    new_elem->tag=tag;
    switch (tag) {
    case 0 : value=atoi(yytext); new_elem->value.int_value=value;t="Int";break;
    case 1 : s=malloc(strlen(yytext)+1);strcpy(s,yytext);
        new_elem->value.str_value=s;t="Str";break;
    case 2 : real=atof(yytext); new_elem->value.real_value=real;t="Real";break;
    case 3 : new_elem->value.bool_value = strcmp(yytext,"FALSE"); t="Bool";break;
    }

    /* Make sure that there is an operator of the base type in the
       operator table */
    if (!is_attr &&(const_id[tag]<0))
        { const_id[tag]=add_id(t);
          if (no_of_opsers>=
              (sizeof(oper_array)/sizeof(oper_array[0])))
              { fprintf(stderr,"Too many operators\n"); exit(99);}
          id_array[const_id[tag]].operator=no_of_opsers;
          oper_array[no_of_opsers].id=const_id[tag];
          oper_array[no_of_opsers].arity=0;
          no_of_opsers++;
        }
    return(no_of_consts++);
}

```

File : test.lex

page 4

```
int add_id(s)
    char *s;
    /* Add an identifier to the list of identifiers */
{ int i;
  char *space;

  /* Linear search through table to see if token is already there */
  for (i=0; i<no_of_ids; i++)
    if (!strcmp(s,id_array[i].name)) return(i);

  /* Not found in table, so table size goes up by one */
  space = (char *) malloc(1+strlen(s));
  if (space == NULL )
    { fprintf(stderr,"Malloc failed\n");exit(99);}
  if (no_of_ids >= (sizeof(id_array)/sizeof(id_array[0])))
    { fprintf(stderr,"Too many identifiers\n");exit(99);}
  strcpy(space,s);
  id_array[no_of_ids].name=space;
  id_array[no_of_ids].operator= -1; /* yet to be determined */
  return(no_of_ids++);
}

yywrap()
{ return(1);}
```

Appendix C

Code for XDR

Chapter 5 describes the transformation of XDR. This appendix contains the IDL specification of the data structure communicated using XDR, as well as the specification of the process that reads the IDL data structure and communicates it using XDR. The data structure communicated by XDR is specified in the C language. The code needed for the transformation is also provided.

C.1 The Specification

This section contains the specification of the IDL data structure communicated using XDR and the data structure communicated by XDR specified in 'C'.

The IDL data structure specification

```
Structure test Root Aval Is
  Aval => first: Cval,
         second: Dval,
         third: Seq Of Integer;
  Bval ::= Cval | Dval ;
  Bval => Enode: Eval;
  Cval => name: String;
  Dval => value: String;
  Eval => number: Integer,
        name: String,
        value: Rational,
        flag: Boolean;

End

Structure proc_inv Root Aval From test Is
  Eval => touched: Integer,
        shared: Integer,
        label_no: Integer;

End

Process writer Inv proc_inv Is
  Target Language C;
  Pre input: test;
  --Post output: XDR;
End
```

The data structure communicated by XDR

```
struct hE {
    enum {ISLABEL=1,ISNODE=2} Etype;
    union {
        int label_no;
        struct hEvalue {
            enum {NOLABEL=1,LABELDEF=2} Etype;
            int label_no;
            struct {
                int number;
                char *name;
                float value;
                int flag;
            } value;
        } value;
    } value;
};
```

```
struct hC {
    char *name;
    struct hE Enode;
};
```

```
struct hD {
    char *value;
    struct hE Enode;
};
```

```
struct intseq {
    int val;
    enum {NULLELEM=1,VALELEM=2} nodetype;
    struct intseq * next;
};
```

```
struct hA {
    struct hC first;
    struct hD second;
    struct intseq third;
};
```

C.2 Input and Output using XDR

This section contains the code that is used to communicate the data structure specified in the previous section using XDR.

File : foo.c

page 1

```

#include <rpc/rpc.h>
#include "foo.h"

bool_t xdr_Evalue(), xdr_intseq();
struct xdr_discrim varm[3]={
    {1, xdr_int},
    {2, xdr_Evalue},
    {100,NULL}
};

struct xdr_discrim warm[3]={
    {1, xdr_void},
    {2, xdr_int},
    {100,NULL}
};

struct xdr_discrim xarm[3]={
    {1,xdr_void},
    {2,xdr_intseq},
    {100,NULL}
};

/* routines to communicate the structure A using XDR */

bool_t xdr_A(xdrs, gp)
    XDR *xdrs;
    struct hA *gp;
{ return (
    xdr_C(xdrs, &(gp->first)) &&
    xdr_D(xdrs, &(gp->second)) &&
    xdr_intseq(xdrs, &(gp->third)));}

bool_t xdr_intseq(xdrs, gp)
    XDR *xdrs;
    struct intseq *gp;
{ if (xdr_enum(xdrs, &(gp->nodetype)))
    if (gp->nodetype==VALELEM)
        return(xdr_int(xdrs, &(gp->val)) &&
            xdr_reference(xdrs, &(gp->next), sizeof(struct intseq),
                xdr_intseq));
    else return(TRUE);
    else return(FALSE);
}

bool_t xdr_C(xdrs, gp)
    XDR *xdrs;
    struct hC *gp;
{ return(xdr_string(xdrs, &(gp->name), 255) &&
    xdr_E(xdrs, &(gp->Enode)));}

```


File : foo.c

page 2

```
bool_t xdr_D(xdrs, gp)
    XDR *xdrs;
    struct hD *gp;
{ return(xdr_string(xdrs, &(gp->value), 255) &&
    xdr_E(xdrs, &(gp->Enode)));}

bool_t xdr_E(xdrs, gp)
    XDR *xdrs;
    struct hE *gp;
{ return(xdr_enum(xdrs, &(gp->Etype)) &&
    xdr_union(xdrs, &(gp->Etype), &(gp->value), warm, NULL));}

bool_t xdr_Evalue(xdrs, gp)
    XDR *xdrs;
    struct hEvalue *gp;
{ return(xdr_enum(xdrs, &(gp->Etype)) &&
    xdr_union(xdrs, &(gp->Etype), &(gp->label_no), warm, NULL) &&
    xdr_int(xdrs, &(gp->value.number)) &&
    xdr_string(xdrs, &(gp->value.name), 255) &&
    xdr_float(xdrs, &(gp->value.value)) &&
    xdr_int(xdrs, &(gp->value.flag)));}
```

C.3 Transformation of the Input Interface

This section contains the code concerned with the transformation of the input interface.

File : main_write.c

page 1

```

#include <stdio.h>
#include <rpc/rpc.h>
#include "foo.h"
#include "writer.h"
int global_label=1;

buildA(idlnode,Cnode)
    Aval idlnode;
    struct hA *Cnode;
{
    buildC(idlnode->first,&(Cnode->first));
    buildD(idlnode->second,&(Cnode->second));
    buildintseq(idlnode->third,&(Cnode->third));
}

buildintseq(idlnode,Cnode)
SEQint idlnode;
struct intseq *Cnode;
{ SEQint tempseq;
  int aval;
  struct intseq *currnode;
  currnode=Cnode;
  foreachinSEQint(idlnode,tempseq,aval)
    { currnode->val=aval;
      currnode->nodetype=VALELEM;
      currnode->next = (struct intseq *) malloc(sizeof(struct intseq));
      currnode=currnode->next;
    }
  currnode->nodetype=NULLELEM;
}

buildC(idlnode,Cnode)
    Cval idlnode;
    struct hC *Cnode;
{
    Cnode->name=StringToChar(idlnode->name);
    buildE(idlnode->Enode,&(Cnode->Enode));
}

buildD(idlnode,Cnode)
    Dval idlnode;
    struct hD *Cnode;
{
    Cnode->value=StringToChar(idlnode->value);
    buildE(idlnode->Enode,&(Cnode->Enode));
}

```

```

buildE(idlnode,Cnode)
    Eval idlnode;
    struct hE *Cnode;
{
    if (idlnode->touched==0)
    { Cnode->Etype=ISLABEL;
      Cnode->value.label_no=idlnode->label_no;
    }
    else {
        Cnode->Etype=ISNODE;
        if (idlnode->shared)
        { Cnode->value.value.Etype=LABELDEF;
          Cnode->value.value.label_no=global_label;
          idlnode->label_no=global_label++;
        }
        else Cnode->value.value.Etype=NOLABEL;
        idlnode->touched=0;
        idlnode->shared=0;
        Cnode->value.value.value.number=idlnode->number;
        Cnode->value.value.value.name=StringToChar(idlnode->name);
        Cnode->value.value.value.value=idlnode->value;
        Cnode->value.value.value.flag=((idlnode->flag)?1:0);
    }
}

markA(node)
    Aval node;
{
    markC(node->first);
    markD(node->second);
}

markC(node)
    Cval node;
{
    markE(node->Enode);
}

markD(node)
    Dval node;
{
    markE(node->Enode);
}

markE(node)
    Eval node;
{
    if (node->touched) node->shared=1;
    node->touched=1;
}

```

File : main_write.c

page 3

```
main(argc,argv)
    int argc;
    char *argv[];
{ XDR xdrs;
  FILE *fp, *fopen();
  struct hA *cstructure;
  Aval idl_struct;
  if (argc!=2) { fprintf(stderr,"Usage:%s <file>\n",argv[0]);exit(99);}
  if ((fp=fopen(argv[1],"r"))==NULL)
    { fprintf(stderr, "Can't open file %s\n", argv[1]);exit(99);}
  xdrstdio_create(&xdrs,stdout,XDR_ENCODE);
  idl_struct=input(fp);
  markA(idl_struct);
  cstructure= (struct hA *) malloc(sizeof(struct hA));
  buildA(idl_struct,cstructure);
  if (!xdr_A(&xdrs,cstructure))
    fprintf(stderr,"XDR failed\n");
}
```

C.4 Transformation of the Output Interface

This section contains code for the transformation of the output interface of XDR.

File : reader.c

page 1

```

#include <stdio.h>
#include <rpc/rpc.h>
#include "foo.h"

bool_t xdr_A();

main()
{
    XDR xdrs;
    struct hA *gp;
    gp = (struct hA *) malloc(sizeof(struct hA));
    xdrstdio_create(&xdrs,stdin,XDR_DECODE);
    if (!xdr_A(&xdrs,gp))
        { fprintf(stderr,"Error: XDR failed\n"); exit(99);}
    output_ascii(gp);
}

output_ascii(gp)
    struct hA *gp;
{ printf("-- structure Aval\n");
  printf("Aval [ \nfirst \n");
  output_C(&(gp->first));
  printf("; second \n");
  output_D(&(gp->second));
  printf("; third <\n");
  output_intseq(&(gp->third));
  printf("]\n#\n");
}

output_intseq(gp)
    struct intseq *gp;
{ struct intseq *currnode;
  currnode=gp;
  while (currnode->nodetype!=NULLELEM) {
    printf(" %d ",currnode->val);
    currnode=currnode->next;
  }
  printf(" >\n");
}

output_C(gp)
    struct hC *gp;
{
  printf("Cval [ name \n");
  printf("\n%s\n" ; \nEnode ".gp->name);
  output_E(&(gp->Enode)); printf("] \n");
}

```

File : main_write.c

page 2

```
output_D(gp)
    struct hD *gp;
{
    printf("Dval [ value \n");
    printf("\'%s\' ; \nEcode ",gp->value);
    output_E(&(gp->Ecode)); printf("] \n");
}

output_E(gp)
    struct hE *gp;
{
    if (gp->Etype==ISLABEL) printf(" L%d^ \n",gp->value.label_no);
    else
    { if (gp->value.value.Etype==LABELDEF)
        printf("L%d: \n",gp->value.value.label_no);
        printf("Eval [ number %d ; \n",gp->value.value.value.number);
        printf("name \'%s\' ; \nvalue %f ; \nflag %d ] \n",
            gp->value.value.value.name, gp->value.value.value.value,
            gp->value.value.value.flag);
    }
}
```


Appendix D

Summary of the Methodology

This appendix presents a summary of the methodology discussed in Chapter 3. This is meant to be a quick reference and does not contain all the details. The methodology consists of the following steps.

Step 1 *Characterise the problem in the dimensions of the problem space described below.*

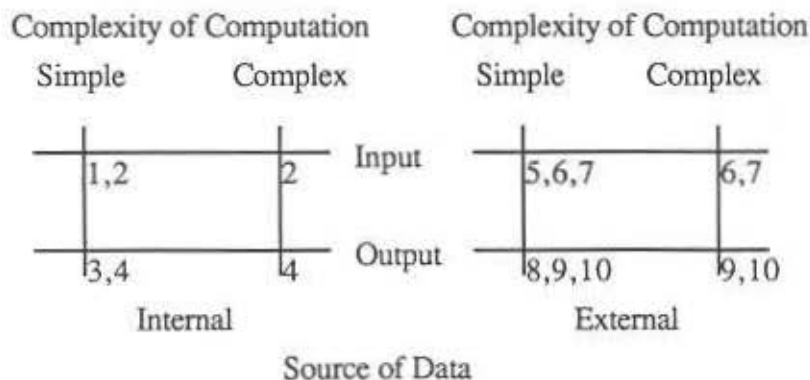
Input or Output Type of interface being transformed, either input or output.

Choice of data Data that may be used to compute the IDL instance (for output) or that may be computed from the IDL instance (for input), either internal or external.

Complexity of Computation Complexity of the computation mentioned above, either simple or complex.

Constraints on the Implementation Availability of the source code for modification, availability of the format of input and output data.

Step 2 *Find the appropriate approach(es) from the mapping.*



Step 3 *Using the metrics evaluate the different approaches. Choose the approach that is best suited following project constraints*

1. Metrics on the transformation.
2. Metrics on the transformed tool.
3. Robustness of the transformed tool.

Step 4 *Implement the approach.*

A brief description of the approaches is presented here.

1. A filter that reads the IDL instance incrementally and computes the input data of the tool.
2. A filter that reads the IDL instance using routines provided by IDLC and computes the input data of the tool from the instance in main memory.
3. A filter that reads the output data of the tool and computes the IDL instance incrementally.
4. A filter that reads the output data of the tool, computes the IDL instance in memory and outputs it using routines provided by IDLC.
5. A subroutine that reads the IDL instance incrementally and computes the internal data structure of the tool.
6. A subroutine that reads the IDL instance into memory using routines provided by IDLC and computes the internal data structure of the tool.
7. Part of internal data structure of the tool is replaced by an IDL structure that is a derivation of the input IDL structure.
8. A subroutine traverses the internal data structure and computes the IDL instance incrementally.
9. A subroutine computes the IDL structure in memory from the internal data structure of the tool.
10. Part of the internal data structure of the tool is replaced by an IDL structure that is a derivation of the output IDL structure.