A Tutorial Introduction to Using IDL

William B. Warren, Jerry Kickenson, and Richard Snodgrass

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27514

1 Introduction

The Interface Description Language (IDL) is a notation for describing the characteristics of data structures passed among a collection of cooperating processes. A tool, the IDL translator, maps these descriptions into code fragments in one of several target programming languages. These code fragments contain declarations of data structures in the target programming language that are functionally equivalent to those described in the IDL specification. The code fragments also define utilities for in-core manipulation and input and output of instances of the data structures. The IDL user writes his programs in terms of the target programming language data declarations and utilities produced by the IDL translator. These programs process instances of the IDL-specified data structures residing on external storage that are cast in terms of an auxiliary language, the ASCII External Representation Language.

This document is a tutorial introduction to using IDL. First, a step-by-step method for using the elementary capabilities of IDL will be given. As each step is examined, the relevant components of the IDL system will be introduced and explained. A complete example of the application of the IDL system will be presented. While IDL supports several target programming languages, the examples in this document use the C programming language. Finally, a brief introduction will be given to the more advanced features of IDL.

Experience with the C programming language and with designing small to medium programs is assumed. It is intended that readers will gain a sufficient understanding of IDL to begin using the IDL system to solve elementary problems.

1.1 History of IDL

The concept of a visible external representation was present in some early compilers designed to be portable across target machines. This concept was refined starting in early 1977 in connection with the Production Quality Compiler-Compiler project of the Computer Science Department of Carnegie-Mellon University under the direction of William Wulf and Joseph Newcomer [Leverett et al. 1980]. At this time, this representation, termed *linear graph notation*, or LG, was defined by Joseph Newcomer and Paul Hilfinger and support software was implemented by Steven Hobbs [Cattell et al. 1980]. The software included a program called REQUIR that accepted a definition of nodes (in LG) and produced a set of data structure definitions in BLISS [Wulf et al. 1971] as well as initialization tables for the LG reader and writer. A family of intermediate languages collectively called TCOL were expressed in LG. One member, for the preliminary Ada language, was $TCOL_{Ada}$ [Brosgol et al. 1980].

A second intermediate representation for Ada programs, AIDA, was developed independently at the University of Karlsruhe [Persch et. al. 1980]. In an effort to merge the best attributes of $TCOL_{Ada}$ and AIDA, the developers of these two representations met in December 1980 and January 1981 to design a new intermediate representation, with DIANA as the result [Goos & Wulf 1981].

Roughly concurrent with the design of DIANA, a successor to LG was being designed by John Nestor and William Wulf, later joined by David Lamb, to address some of the problems that had been identified in LG. This successor, based on the concept of representation- and language- independent data definition, evolved into IDL. This language was further refined at the joint meeting (one major change being the syntactic form suggested by Gerhard Goos) and was used to express DIANA. The original IDL system consisted of the definition of the Interface Definition Language for describing data structures, processes, and assertions, and the definition of another language, called the ASCII External Representation Language, for representing instances of IDL-specified data structures on external storage. Later that year, researchers at Carnegie-Mellon University had implemented a minimal translator for IDL. Within another year, a formal definition of the IDL language, the external representation language, and the assertion language had been completed [Nestor, et al. 1982].

In May of 1983, David Lamb published his Ph.D. dissertation [Lamb 1983] in which he presented the results of his investigations into the practicality of using IDL as a tool for connecting the components of large software systems. His work focused on developing a design for a translator of the full IDL language, including the assertion language. He demonstrated that a translator for the base language was feasible and could be made acceptably efficient. He also formulated the design of an assertion checker, and investigated efficiency issues in that context.

DIANA has been used in compilers implemented at Bell Labs, Burroughs, University of California, Berkeley [Zorn 1985], Intermetrics, University of Karlsruhe, Rolm, and SofTech [Butler 1983]. Tartan Labs has used IDL as the basis for most of its tools. These efforts have generally involved proprietary translators and runtime libraries. Most recently, in 1985 members of the SoftLab Project at the University of North Carolina at Chapel Hill completed an implementation of an IDL translator and of a set of IDL development support tools running on Unix [Snodgrass 1985]. The first version supported C as a target programming language; work continues on supporting mappings of IDL specifications to other target programming languages and on developing additional support tools.

1.2 Purpose of IDL

IDL is particularly useful for describing *graph-structured* data that is passed between a collection of cooperating processes. Examples of graph-structured data are stacks, queues, lists, trees, graphs, etc. A good example of the type of system to which IDL is best applied is a compiler such as that shown in block form in Figure 1. In this diagram, the ovals represent phases of the compiler's process and the boxes represent data; both ovals and boxes contain labels to suggest their roles.

Each phase of the compiler except for the first and last receives a rather complex data structure from the previous phase, alters that data structure in some way, and passes the transformed data structure to the next phase. IDL is an ideal tool for describing the data structures passed from one phase to the next.

1.3 Model of a Process

IDL assumes a particular model of a process (phase, tool) of a software system. In this model, shown in Figure 2, each process reads one or more input data structures, termed *instances*, into main memory, using the IDL reader, stored in language-specific runtime data structures. Utilities (e.g., macros, functions) are provided to manipulate this structure. The user-supplied algorithm uses information from the input instance(s) to create new instance(s), which are written out using the IDL writer. In Figure 1, there are four processes, each taking one input instance and generating one output instance. The process is described in an *IDL specification*: precise descriptions are given of the structure of each of the instances read or written. From this specification, a tool, the *IDL translator*, generates type declarations, as well as readers and writers, in the target language. The algorithm itself must be supplied by the user. While the algorithm may read and write other data or text, such behavior is not modeled by IDL.

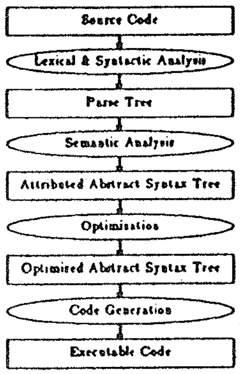


Figure 1 Block Diagram for a Typical Compiler

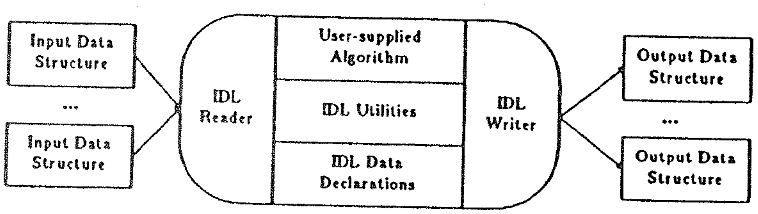


Figure 2 Model of a Process

1.4 Benefits of IDL

Using IDL simplifies the development of complex software systems. Designing the system is easier because IDL permits a higher level of data abstraction than that provided by most programming languages. Abstract data types such as sets and sequences for any type, complete with all necessary data declarations and data manipulation routines, are supported by IDL. Consequently, the user has the opportunity to more naturally express his algorithm in terms of these abstractions without becoming mired in implementation detail.

Building the system is faster because there is less code to write and debug. Since a large portion of the data manipulation utility routines are provided bug-free by the IDL system, the user must write and debug only the code for his algorithm. This in turn makes for fast prototyping of the system which permits an iterative design approach.

Finally, the software system that has been developed using the IDL system is better documented and thus easier to maintain. Data structures specified with IDL are documented by their specifications. The IDL-supplied data manipulation routines are documented in the IDL system documentation.

1.5 IDL Variants

Implicit in any implementation of IDL are assumptions about how the instances of IDL structures are used. These assumptions may restrict the operations slightly in order to achieve reasonable efficiency. Hence, each implementation unavoidably defines a variant of the specification language that may differ from that supported by another implementation. The language described in this document is the one supported by the SoftLab tools implemented at the University of North Carolina at Chapel Hill.

1.6 Conventions Used in This Document

Words or phrases that denote important concepts will be printed in *italics* on first appearance. In the programming language and IDL specification examples appearing in the running text, all IDL, C, and Pascal reserved words as well as required punctuation appear in *`slanted typewriter`* font. In addition, user input appears in **bold Roman** font (such uses will be infrequent). User-defined identifiers and file names are shown in `typewriter` font. Comments appear in *italics*. A SMALL CAPITALIZED font is used for program names (e.g., TREEPR). In all IDL specifications, class names start with an upper case letter and node and attribute names are start with a lower case letter.

All syntactic definitions are given in an extended version of the Backus-Naur Form (BNF). Angle brackets ("⟨⟩") surround the name of a non-terminal. Braces ("{}") are used to group elements of a production; a trailing asterisk ("}* ") indicates zero or more occurrences; a trailing plus ("}+ ") indicates one or more occurrences; a trailing question mark ("}? ") indicates an optional item. Special characters that are terminals such as a semicolon are identified by prefixing them with a single quote mark (e.g., " ';").

2 The Steps in Using IDL

Figure 3 presents the flow of using the IDL system. Once again, the ovals represent processes and the boxes represent data. A line from a box to an oval indicates that the data is read by the process; a line from an oval to a box indicates that the data was produced by the process.

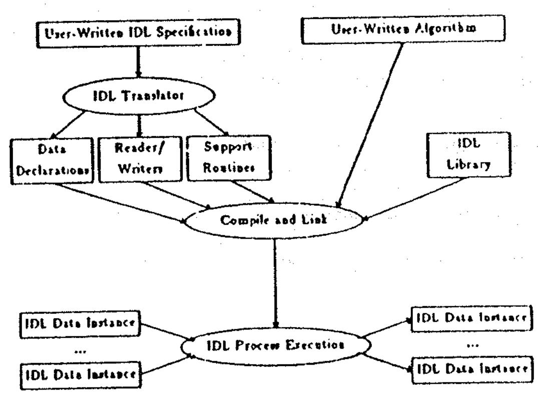


Figure 3 Steps in Using the IDL System

The user of the IDL system first writes specifications for his data structures in the Interface Description Language. In the next step, the user translates those specifications with the IDL translator into a set of data declarations in the target programming language that expresses the same functionality. The IDL translator also produces a set of run-time support routines tailored for manipulating the target-language versions of his data structures. The user then proceeds to program his algorithm in the selected target programming language in terms of the data declarations produced by the IDL translator. While programming, he makes use of the tailored set of run-time support routines to allocate new instances of data structures, to maintain sets and sequences composed of them, and to read and write them from external storage. The user then compiles and links the code for his algorithm along with the IDL-produced data declarations and run-time support routines to produce an executable program. Finally, the user executes this program to process instances of data structures that are represented externally in the ASCII External Representation Language.

This chapter considers each of these steps separately. As a phase is discussed, the relevant parts of the IDL system are explained. The next chapter examines each step once again in the context of a second example.

2.1 The IDL Specification

Writing a specification of the collection of cooperating processes that compose the system being developed and of the data structures those processes share is the first step in using the IDL system. Figure 4 identifies this portion of the whole process of using the IDL system. The specification is written in the Interface Description Language. The specifications for one or more data structures and one or more processes constitute a complete specification.

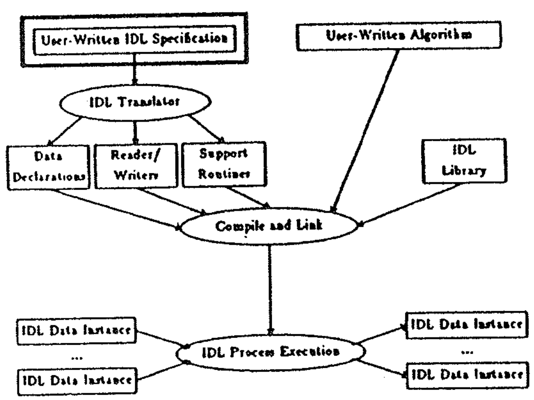


Figure 4 The Specification Step

2.1.1 *Specifying Data Structures*

The fundamental data structure building blocks of IDL are nodes and classes. Nodes and classes are organized into named collections called structures.

Nodes A *node* is a named collection of zero or more named values called *attributes* that the user wishes to treat as a unit. Attributes actually hold the data values; nodes are a grouping device. Nodes are analogous to records found in many programming languages; their attributes are analogous to the fields of a record. It is important to note that nodes need not actually *be* records—they just act like they are. In particular, the values for attributes may be stored outside the node, or may not be stored at all, instead being recomputed whenever the attribute is accessed. IDL is an *abstract specification* capable of being mapped onto a myriad of representations.

A declaration for a node consists of the name of the node followed by a *node production operator*, a "=>", and then a list of zero or more comma-separated attribute-type pairs terminated with a semicolon. The syntax for declaring a node is given below.

(node name) '=> {(attribute name) ': (type) {', (attribute name) ': (type)}* ';

A node may have any number of attributes limited only by the implementation of the IDL translator. The ordering of the attributes within a node is not significant. Names for nodes and attributes and for any other named IDL objects are contiguous sequences of letters, digits, and the underscore character "_". The first character must be a letter or an underscore; case is significant. The length of a name is limited only by the implementation of the IDL translator. All characters in a name are considered significant. All IDL keywords (introduced below) are reserved and must be spelled and capitalized exactly as they appear. The various parts of a declaration may be separated by any amount of white space, i.e., blanks, tabs, and newlines.

An attribute's *type* specifies the domain of values that the attribute can hold. IDL provides four *basic types* and two kinds of *structured types*. The IDL keywords `Boolean`, `Integer`, `Rational`, and `String` name the basic types. `Booleans` can have values `True` and `False`. `Integers` are theoretically unbounded; all implementations have a practical limit. `Rationals` are technically fractions with integral numerator and denominator; this definition encompasses conventional floating point representations such as Pascal's `real` and C's `float`, as well as the fixed point types of PL/1 and Ada. A `String` is a bounded sequence of characters. Some languages such as C support them directly; in the others, such as Pascal, they are often represented as records. The following node declaration specifies six attributes: a `name` attribute of type `String`, a `state_code` attribute of type `Integer`, an `address` attribute of type `String`, an `active` attribute of type `Boolean`, a `customer_number` attribute of type `Integer`, and a `balance` attribute of type `Rational`.

```
state_customer => name            : String,
                  address         : String,
                  customer_number : Integer,
                  active          : Boolean,
                  balance         : Rational,
                  state_code      : Integer;
```

Figure 5 A Node Type Declaration

IDL provides two kinds of structured types, named by the keywords `Set Of` (type) and `Seq Of` (type). A `Set Of` (type) is an unordered collection (set) of objects of (type). Here, (type) stands for any valid attribute type, other than sets or sequences. Duplication of objects is not permitted within a set. A `Seq Of` (type) is an ordered collection (sequence) of objects of (type). Duplication of objects *is* allowed in a sequence. The IDL system automatically supplies a collection of run-time support routines to the user to manipulate sets and sequences in accordance with their expected behavior.

The `bill_list` node contains one attribute, a sequence of nodes of type `bill`:

```
bill_list => list : Seq of bill;
```

The IDL translator generates over a dozen support routines sequences of bills, including the predicate `inSEQbill`, which determines whether a particular bill is in a sequence, `appendfrontSEQbill`, and `retrievefirstSEQbill`. As with the basic types and nodes, there are many possible representations for sequences, from the mundane ones such as arrays and linked lists to the more interesting ones like threaded sequence, where an attribute in the node is used as a pointer (perhaps abstractly, say through a hash table) to the next node of the sequence. Whatever the representation, the support routines are provided for that representation.

Nodes may be declared without any attributes. The following definition of `local_sales_tax` is an example of such a node.

```
local_sales_tax => ;
```

We will see the usefulness of such nodes later.

In addition to the four basic and two structured types supplied by IDL, an attribute may have as a type a node or class (defined below) that the user has declared in the same structure. As example of a declaration of a node with an attribute whose type is anothe node:

```
bill    =>  billee :  state_customer,
            amount :  Rational;
```

In this example, the `billee` attribute of the `bill` node has type `state_customer` which is a node.

An attribute with a type that is the name of a node is a reference to that node. Nodes may be self-referential through their attributes, such as in the node `binary_tree` declared below.

```
binary_tree => name         :  String,
               left_branch  :  binary_tree,
               right_branch :  binary_tree;
```

In this example, the `binary_tree` node has two attributes, the `left_branch` and the `right_branch` that refer to nodes of the same type, i.e., `binary_tree`.

Attributes within the same node must have different names. Attributes in different nodes may share identical names without conflict. Identically-named attributes in different nodes may even have different types.

Classes A *class* can hold a reference to one of a set of nodes or other classes. The elements of the set of nodes or other classes that a class can refer to are called its *members*. A class is used to state some common aspect of its members, such as the fact that all contain a particular attribute. As with nodes, there are a great many ways to represent classes. Possible implementations of IDL classes are Pascal variant `records`, C `unions`, and Simula and Smalltalk `classes`.

A declaration of a class consists of the name of the new class followed by a *class production operator*, a "::=", and then a list of one or more node or other class names separated by alteration signs, "|", and terminated with a semicolon.

⟨class name⟩ '::= {⟨class name⟩ | ⟨node name⟩} {'| {⟨class name⟩ | ⟨node name⟩} }* ';

There is no significance to the ordering of the nodes and classes on the right hand side of a declaration. Restrictions on classes will be discussed later.

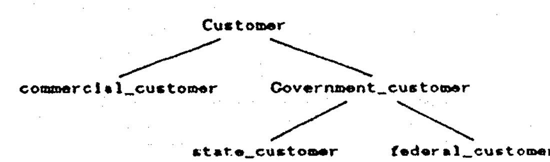
In Figure 5 a class named `Customer` is declared to have two members, a `commercial_customer` node and a `Government_customer` class. The `Government_customer` class is declared to have two members, a `state_customer` node, and a `federal_customer` node.

```
Customer            ::= commercial_customer| Government_customer;

commercial_customer => name               :  String,
                       industry_code      :  Integer,
                       address            :  String,
                       customer_number    :  Integer;

Government_customer ::= state_customer | federal_customer;

state_customer      ::= name              :  String,
                       state_code         :  Integer,
                       address            :  String,
                       customer_number    :  Integer;

federal_customer    => name               :  String,
                       agency_code        :  Integer,
                       address            :  String,
                       customer_number    :  Integer;
```

Figure 5 Class Type Declarations

The members of a class that are listed in its declaration are said to be *direct class members*. In the example above, the `commercial_customer` node and the `Government_Customer` class are the direct class members of the `Customer` class. These are not the only members of the `Customer` class, however. Through a process called *class membership inheritance*, the members of the `Government_Customer` class are also considered to be members of the `Customer` class. This process of membership inheritance repeats indefinitely. The members of a class that are inherited through other classes in this manner are said to be *indirect class members* of the class.

The following figure portrays the class membership forest for the example in Figure 6.



The interior branches are the `Customer` and `Government_customer` classes and the leaves are the `commercial_customer`, `state_customer`, and `federal_customer` nodes. The `Government_customer` class in this figure has the `state_customer` and `federal_customer` nodes as its only members.

There are two rules that restrict the members of a class. First, every class must have at least one member. Second, no class may be a direct or indirect class member of itself. These rules taken together imply some properties of classes. Specifically, class graphs are arbitrary multiple-rooted directed acyclic graphs with IDL node types as leaves.

The class concept in IDL is similar to that of the same name in Simula-67 [Birtwistle, et al. 1973], Smalltalk [Goldberg & Robson 1983], or C++ [Stroustrup 1986]. They are similar in that these languages all support the definition of a hierarchical collection of classes (IDL and Smalltalk allow multiple hierarchies) and that subclasses in these languages inherit attributes defined in superclasses. They differ in that IDL deals only with data and not with computations, whereas the other languages allow procedures to be attached to classes. A second difference is that IDL emphasizes the automatic construction of readers and writers of structure instances; the other languages require the user to implement the readers and writers.

Node and Class Declaration Writing Details

As stated above, an attribute of a node may have a type that is a class declared elsewhere. An example of this is shown below.

```
list_of_customers => list:  Seq Of Customer;
```

The `list` attribute of the node `list_of_customers` has type ***Seq Of* Customer** where **Customer** is a class.

When all of the direct or indirect class member nodes of a particular class share one or more attributes, the user should assign the shared attributes to all of the direct or indirect class member nodes simultaneously through the class rather than making assignments of the common attributes to each of those nodes individually. This allows the user to make a clear statement of the similarities among the member nodes of a class and helps him to avoid unintentional differences.

The syntax for assigning attributes to all direct and indirect class member nodes through a class is given below.

⟨attribute name⟩ ':' ⟨type⟩ {',' ⟨attribute name⟩ ':' ⟨type⟩ }* ';'

The syntax is identical to that for a node declaration with the ⟨node name⟩ replaced by the name of the class. For example, the declarations in Figure 6 could have been written as in Figure 7.

```
Customer            ::= commercial_customer | Government_customer;
Customer            =>  name              :  String,
                        address           :  String,
                        customer_number   :  Integer;

commercial_customer =>  industry_code     :  Integer;

Government_customer ::= state_customer | federal_customer;
state_customer      =>  state_code        :  Integer;
federal_customer    =>  agency_code       :  Integer;
```

Figure 7 Defining Attributes in Classes

In the second declaration of this figure, the **name**, **address**, and **customer_number** attributes are assigned to **commercial_customer**, **state_customer**, and **federal_customer** nodes through the **Customer** class. Note that these attributes propagated through the **Government_customer** class to the **state_customer** and **federal_customer** nodes. In this form, it is easy to see how the **commercial_customer**, **state_customer**, and **federal_customer** nodes are alike and how they differ. This second form is also six lines shorter than that of Figure 6.

Two subtle semantic differences exist between the specifications in Figures 6 and 7. If another node is added to the **Customer** class, it will automatically inherit the **name**, **address**, and **customer_number** attributes in the latter specification, but not in the former. More importantly, the assertion language and some target language implementations of IDL, including the C interface to be discussed shortly, will not allow attributes of variables of a class type to be accessed or shared unless the attribute is especially associated with that class, or with a class of which it is a member.

IDL permits the user to declare a node or class all at once or to split its declaration into several parts. The effect of a multi-part declaration is cumulative. The **state_customer** node declared in Figure 5, for example, could have been declared in two different places as in the following example.

```
state_customer =>  name              :  String,
                   state_code        :  Integer,
                   address           :  String;

-- Zero or more intervening node or class type declarations.

state_customer =>  active            :  Boolean,
                   customer_number   :  Integer,
                   balance           :  Rational;
```

declaration of the **state_customer** node is the union of both declaration "pieces". This permits the user to group the parts of a node or class declaration however he wishes for the sake of clarity.

Structures declarations in an IDL specification are grouped into named collections called ***structures***. A declaration of a structure starts with the keyword ***Structure*** followed by the structure's name, then by the designation of a node or class as its root introduced by the keyword ***Root***, and then by a list of the one or more node and class declarations that comprise the structure between the keywords ***Is*** and ***End***.

Structure ⟨structure name⟩ ***Root*** {⟨class name⟩ | ⟨node name⟩} ***Is***
{⟨node or class declaration⟩}+
End

Within a structure, the ordering of the node and class declarations is not significant; node and class declarations may be listed without regard to forward references. Every structure must contain at least one declaration. Furthermore, as a consequence of the rules for classes, every structure must contain at least one node declaration.

An example of a structure specification is shown in Figure 8.

```
-- Specification for the transactions structure.
Structure transactions Root transaction_list Is
   transaction_list  =>  list              :  Seq Of Transaction;

   Transaction       ::= credit | debit;
   Transaction       =>  customer_number   :  Integer,
                         date              :  Integer,
                         amount            :  Rational,
                         tax_status        :  Set Of Tax_code;

   credit            =>  ;
   debit             =>  ;

   Tax_code          ::=   local_sales_tax
                         | state_sales_tax
                         | federal_sales_tax;
   local_sales_tax   =>  ;
   state_sales_tax   =>  ;
   federal_sales_tax =>  ;
End
```

Figure 8 A Structure Declaration

A structure named **transactions** is declared in this example consisting of two classes, **Transaction** and **Tax_code**, and six nodes, **transaction_list**, **credit**, **debit**, **local_sales_tax**, **state_sales_tax**, and **federal_sales_tax**. The **credit** and **debit** nodes have the same attributes, **customer_number**, **date**, and **amount**, which have been inherited through the **Transaction** class. The **local_sales_tax**, **state_sales_tax**, and **federal_sales_tax** nodes are examples of unattributed nodes. The **transaction_list** node has been designated as the root of the structure.

Structure Writing Details Just as the node defines a scope for the names of its attributes, the structure defines a scope for the names of its nodes and classes. The names of different nodes and classes within the same structure must be different. Only those nodes and classes that have been declared within a structure may be referenced as attribute types within that structure.

All nodes and classes must be *reachable* from the root node or class of the structure. It must be possible to trace a path of any number of steps from the root node or class to all other nodes and classes declared in the structure. A path is traced through a node to other nodes and classes via the attributes of that node that have a type that is another node or class. A path is traced through a class to other nodes and classes via its direct class member nodes or classes. For example, in Figure 8, the `Transaction` class is reachable directly from the `transaction_list` through the type of its `list` attribute. The `credit` and `debit` nodes are then indirectly reachable from the `transaction_list` node through the `Transaction` class.

Note how the concept of a path in this context differs from that discussed in the section on classes with regards to class membership. In determining class membership, only those paths that we can trace through direct and indirect class membership are of interest. For deciding reachability, however, one may consider paths traced through either attribute types or class membership or through a combination of both.

The reachability requirement guards against spelling mistakes, missing class membership declarations, and missing attribute declarations. Without this check, some of these errors lead to structures that make no sense.

At run-time, the root of a structure serves much like the root of a tree. Actual instances of the IDL-specified data structures, whether internal to a program or on external storage, consist of an instance of the root node or class with instances of one or more of the other nodes or classes attached. The attachments of the other nodes or classes to the root are made through attribute types and/or class membership. For example, an actual instance of the `transactions` structure specified in Figure 8 would consist of a single instance of the `transaction_list` node with an attached list of zero or more members of the `Transaction` class, either `credits` or `debits`. Generally, as in this case, the node or class that is named as the root is one that the user views as a backbone of the data structure.

At this point, it may appear to the reader to be a difficult task to determine from a specification which objects within a structure are nodes and which are classes if a specification has been written making use of attribute assignments through classes and multi-part node and class declarations. This determination can be made quite simply, however. The classes within a structure are those objects that appear to the left of at least one class production operator no matter how many times they may appear to the left of node production operators. The nodes within a structure are those objects that appear to the left of one or more node production operators only. Appearances of classes to the left of node production operators, if there are any, are attribute assignments to its member nodes. As an aid to the reader, the class names in all of the IDL specifications throughout this document are capitalized and the node and attribute names appear in lower case.

2.1.2 Specifying Processes

A *process* is the IDL model for a computation. An instance of a process reads and writes instances of IDL-specified data structures to and from external storage through a collection of ports. Each process has a master data structure called the invariant that is the union of all data structures used in that process.

Ports A *port* is an association between an IDL-specified data structure and a name for the IDL-supplied implementation of the routines for reading or writing that structure to and from a process. There are two kinds of ports; *Pre* ports for input of instances of the associated data structure, and *Post* ports for output. The declaration of a port consists of the type of port, *Pre* or *Post*, followed by a comma separated list of port name, structure name pairs separated by colons. The list is terminated with a semicolon.

Pre ⟨port name⟩ ':' ⟨structure name⟩ {',' ⟨port name⟩ ':' ⟨structure name⟩}* ';'

Post ⟨port name⟩ ':' ⟨structure name⟩ {',' ⟨port name⟩ ':' ⟨structure name⟩}* ';'

Processes Processes are analogous to C programs. An instance of a process reads zero or more instances of IDL-specified data structures from external storage or another process, performs some computation on its inputs, and writes out instances of zero or more new or transformed IDL-specified data structures to external storage or another process. Figure 2 illustrated this model of a process. The declaration specifies the name of the process and the names of the collection of IDL-specified data structures read and written by an instance of the process. The IDL declaration does not describe the manipulation of the data structures made during the computation.

A declaration of a process starts with the IDL keyword *Process* followed by the process' name, then by a list of the declarations of the ports used by the process that begins with the IDL keyword *Is* and ends with the IDL keyword *End*.

Process ⟨process name⟩ *Is*
    {⟨process statement⟩}+
End

An example of a process specification is shown in Figure 9.

```
-- Specification for the billing process.
Process billing Is
    Pre customers_in     :  customers;
    Pre transactions_in  :  transactions;
    Post customers_out   :  customers;
    Post bills_out       :  bills;
End
```

Figure 9 A Process Declaration

In this example, a process named `billing` is declared. The `billing` process has two *Pre* ports named `customers_in` and `transactions_in` that read instances of the `customers` and `transactions` data structures, respectively, and two *Post* ports named `customers_out` and `bills_out` that write instances of the `customers` and `bills` data structures, respectively.

Within a process declaration, each port name must differ from all other port names and structure names referred to within that process. There may be any number of *Pre* or *Post* port declaration sequences within the same process declaration as in Figure 9. The order of port declarations within a process is not significant.

The user should note that an instance of an IDL-specified process may do I/O other than that specified in the process specification. The process specification only captures the I/O behavior of the process concerning reading and writing instances of IDL-specified data structures.

2.1.3 A Complete Specification

A complete IDL specification is shown in Figure 10. This specification describes a single proces, `billing` that reads in instances of two data structures, `customers` and `transactions` through the `customers_in` and `transactions_in` ports respectively, creates a set of bills for the customers described in an instance of the `bills` data structure, modifies the instance of the `customers` data structure, and writes out instances of the new `bills` data structure and the modified `customers` data structure through the `bills_out` and `customers_out` ports respectively.

```
-- Specification for the customers structure.
Structure customers Root customer_list Is
    customer_list         => list               : Seq Of Customer;

    Customer              ::= commercial_customer | Government_customer;
    Customer              => name               : String,
                             address            : String,
                             active             : Boolean,
                             customer_number    : Integer,
                             balance            : Rational;

    commercial_customer => industry_code      : Integer;
    Government_customer ::= state_customer | federal_customer;
    state_customer        => state_code         : Integer;
    federal_customer      => agency_code        : Integer;
End

-- Specification for the transactions structure.
Structure transactions Root transaction_list Is
    transaction_list      => list               : Seq Of Transaction;

    Transaction           ::= credit | debit;
    Transaction           => customer_number    : Integer,
                             date               : Integer,
                             amount             : Rational,
                             tax_status         : Set Of Tax_code;

    credit                => ;
    debit                 => ;

    Tax_code              ::= local_sales_tax
                              | state_sales_tax
                              | federal_sales_tax;

    local_sales_tax       => ;
    state_sales_tax       => ;
    federal_sales_tax     => ;

End

-- Specification for the bills structure.
Structure bills Root bill_list Is

    bill_list             => list               : Seq Of bill;
    bill                  => billee             : Customer,
                             amount             : Rational;

    Customer              ::= commercial_customer | Government_customer;
    Customer              => name               : String,
                             address            : String,
                             customer_number    : Integer;
    commercial_customer => industry_code      : Integer;

    Government_customer ::= state_customer | federal_customer;
    state_customer        => state_code         : Integer;
    federal_customer      => agency_code        : Integer;
End

-- Specification for the billing process.
Process billing Is

    Pre customers_in      : customers;
    Pre customers_in      : customers;
    Pre transactions_in   : transactions;
    Post customers_out    : customers;
    Post bills_out        : bills;

End
```

Figure 10 A Complete Specification

Note that this process uses several different data structures. It reads in two data structures, creates a new data structure, and writes out the new data structure and a modified version of one of the data structures it read in. The other data structure that was read in is discarded after being processed. Many other combinations of input and output behavior of structures is possible. In the next chapter we will see an example of a process that reads in a single data structure, modifies it, and writes out the modified data structure.

Also note how the elements of this specification are laid out on the page. In the structure declarations, the lines beginning with the keywords ***Structure*** and **End** bracket the indented lists of the nodes and classes that make up each structure. Within the lists of node and class declarations, the node and class production operators are aligned vertically followed by the lists of the constituent node attributes and class members also aligned vertically. The process declarations and their lists of port declarations are laid out in a similar fashion. While the IDL translator ignores all white space, consistent indenting will aid in making a clear specification.

2.1.4 Specifying Assertions

The IDL Assertion Language is a sublanguage of IDL. It permits the user of IDL to make assertions about the IDL structures he has written and the values of attributes within those structures. A program called IDLCHECK can then automatically check the validity of these assertions on particular structure instances.

This section is a tutorial introduction to using the assertion language and IDLCHECK. Basic features of the assertion language will be described, along with examples of their use. The process by which the assertions were composed will be explained.

Purpose of Assertions The assertion language is meant to provide both specification and verification facilities to IDL.

As a specification language, the assertions provide a means for the programmer to precisely specify what it is he wants to be true in his data structures. Using the assertion language facilitates communication between members of a programming team or between future and present programmers. Maintenance of large programs is eased, since the maintainer may look at the assertions to ascertain exactly what is supposed to be accomplished, rather than attempting to figure this out by directly reading code or comments (which are often sparse, incomplete, and imprecise). In addition, writing assertions serves to hone the programmer's thinking about a particular problem, since to write meaningful assertions requires a thorough understanding of the problem. Finally, the assertions provide a guide to the writing of the actual code. Assertions are given in terms of the IDL structures. Programming in the IDL system involves manipulating these structures. Thus, well written assertions state precisely what modifications of the IDL structures must be accomplished by the code.

As a verification language, assertions provide a debugging aid. Once the assertions are written, instances of data structures can be checked automatically. No longer does the programmer have to manually scan through pages of output to ensure that there were no errors. Errors will be found and reported.

Figure 11 illustrates where the assertion checker tool fits in with the other IDL system tools. Rectangles denote data, ellipses denote executable code, and arrows denote input and output of IDL instances.

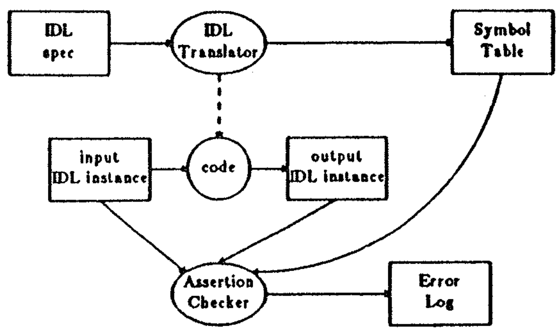


Figure 11 Assertion Checker Tool

In the above figure, assertions are placed in the IDL specifications (*IDL spec*). The user writes some of the code (in particular, the implementation of the algorithm). Everything else is generated automatically. The translator IDLO produces much of the code and the analyzed specification. IDLOHEOK generates the *Error Log*, a report of any user assertions found to be false.

The example specification in Figure 10 will be used to illustrate use of the assertion language.

Basic Assertions With Quantifiers Assertions always begin with the keyword **Assert**. The simplest assertions to make concern the values of specified attributes within a structure. These assertions may appear anywhere within the structure to which they refer, although it is often clearer to group all assertions together. Assertions can be made concerning a property about a single object or a property that should hold for many objects. For example, to assert in the customers structure that the customer list is not empty, we would write:

```
Assert Size(Root.list) ~= 0;
```

Root refers to the root of the structure in which the assertion appears. In this case, the root of the customers structure is of type customer_list. Thus, Root.list is a sequence of Customer. The assertion states that the size of this sequence is not zero.

Usually one wishes to make an assertion about all objects of a certain type. This is done using *quantifiers*. There are two quantifiers in the language. The **ForAll** variant asserts that the body is true for all objects of the specified type. For example, we might wish to make some assertions about the customers structure. First, all customer numbers should be greater than or equal to one. A negative customer number would signify an input error. We would assert this as:

```
Assert ForAll C In Customer Do C.customer_number >= 1 Od;
```

The name C is an *iterator* which will take on successive values of the specified object type, which here is Customer. The body of the quantifier states that the customer_number attribute of C (C is of type Customer) is greater than or equal to one. The period (".") indicates the extraction of an attribute. (This parallels the "=>" symbol in the IDL specification.) Since the **ForAll** variant is used, we are asserting that the customer_number attribute of all objects of type Customer have the described property.

The Exists variant asserts that the body is true for at least one object of the specified type. For example, to assert that at least one customer is active, we would write:

```
Assert Exists C In Customer Do C.active = True Od;
```

The body of a quantifier, whether **ForAll** or **Exists**, is a boolean expression. Despite the use of the Do...Od form, nothing is *done*. We are asserting what must be true. Note also that the type of the iterator, specified immediately after the keyword *In*, must be a valid type in the IDL specification. A valid type is any node or class name (but not attribute names) or a basic type (Integer, Rational, String, Seq Of (type), or Set Of (type)).

Boolean operators such as *And*, *Or*, and *Not* may be used. For example, we wish to assert that the state_code of every state_customer is between 1 and 50, inclusive. We could write:

```
Assert ForAll sc In state_customer Do sc.state_code >= 1 Andsc.state_code <= 50 Od;
```

Nesting of quantifiers is allowed. For example, we wish to assert that no two distinct customers have the same customer number, to avoid confusing bills and credits. We would assert as follows:

```
-- No duplicate customer numbers
Assert ForAll c1 In Customer Do
        ForAll c2 In Customer Do
                If c1 ~= c2 Then
                        c1.customer_number ~= c2.customer_number
                Else True Fi
        Od Od;
```

The symbol "~=" means "not equal to". The two iterator names must be distinct when using nested quantifiers. Here the names used are c1 and c2. *Od* appears twice at the end. Each occurrence of a quantifier must be ended by its own *Od*.

Similar assertions on attribute values might be made in the transactions structure and the bills structure. For instance, we could assert in the transactions structure:

```
Assert ForAll t In Transaction Do t.customer_number >= 1 Od;
```

In the bills structure, we might wish to be certain that the amount of all bills is positive. We could then assert in the bills structure:

```
Assert ForAll b In bill Do b.amount > 0 Od;
```

The assertion language supplies a few built-in functions which are useful. The function Size was used above. It returns the number of objects in a set, sequence or collection (described below) or the number of characters in a string. Other functions will be introduced below. String constants are characters enclosed within quotes. For example, we wish to be certain that a very important customer named IBM is in our customer list. We would then assert:

```
Assert Exists C In Customer Do C.name = "IBM" Od;
```

Assertions within Processes The above assertions concern particular structures. They are thus placed inside the structure to which they refer. It is also possible to make assertions within process specifications. In this way, one can make assertions about the relation of output structures to input structures.

For example, in the process billing defined in section 2.1.2, there is an instance of the customers structure read in and an instance of the customers structure written out. The information within these instances should remain unaltered, except perhaps for the balance attribute of Customer. We could assert this as follows (within the process specification):

```
-- Customers output match customers input.
Assert ForAll c_in In customers_in:Customer Do
        ForAll c_out In customers_out:Customer Do
                If c_in.customer_number = c_out.customer_number Then
                        c_in.name = c_out.name And
```

```
                c_in.address = c_out.address And
                c_in.active = c_out.active
            Else True Fi
    Od Od;
```

The specified iterator types in the above quantifiers are prefixed with a port name followed by a colon. The prefix specifies the name of the port associated with the structure instance the assertion is referring to. In the above example, the first quantifier is referring to all objects of type Customer in the structure associated with the port named customer_in. The second quantifier is referring to all objects of type Customer in the structure associated with the port named Customer_out. These structures must both contain the type specified (in this case Customer). The use of these *port prefixes* is allowed only within a process specification, since structure specifications do not have ports. Clearly, a port with the name specified in the prefix should exist if the assertion is to make sense.

In the last example the customer_number and balance attributes were not mentioned. This is because the *If* condition already guarantees that the customer_number attributes are equal, and the balance attribute may be legitimately altered due to transactions being processed. In addition, the industry_code, state_code, and agency_code attributes were not mentioned. These should remain unaltered also, but they are not attributes of the entire class Customer. They are attributes of particular node members of that class. Thus statements about them cannot be made within the context of the entire class. For example, to assert:

```
-- Bad Assertion!
Assert ForAll c In Customer Do c.customer_number > 0 And c.industry_code > 0 Od;
```

would result in a compile-time typing error, because c is of type Customer, which does not necessarily have an industry_code attribute. If one wishes to make an assertion concerning such an attribute, one must write an assertion specifically targeted toward that class of node containing the attribute. For example, we could assert:

```
Assert ForAll cc_in In customers_in:commercial_customer Do
        ForAll cc_out In customers_out:commercial_customer Do
                If cc_in.customer_number = cc_out.customer_number Then
                        cc_in.industry_code = cc_out.industry_code
                Else True Fi
        Od Od;
```

Since commercial_customer is a member of the Customer class, we can refer to the customer_number attribute of cc_in and cc_out which are of type Customer. Since we have specified cc_in and cc_out as type commercial_customer, we can now also refer to an industry_code attribute. Similar assertions could be made about the state_code for objects of type state_customer and the agency_code for objects of type federal_customer.

As another example, we can assert that every transaction refers to an actual customer.

```
-- Each transaction refers to an actual customer
Assert ForAll t In transactions_in:Transaction Do
        Exists c In customers_in:Customer Do t.customer_number = c.customer_number Od
        Od;
```

Assertions can get more complicated than the examples above. In order to simplify the formation of these assertions, the assertion language allows one to create definitions, and then use these definitions in assertions.

Value Returning Definitions The simplest kind of definition returns a value. For example, we wish to assert that the balance of a customer on output from our billing process is equal to the balance the customer had on input plus any credit received through transactions. Essentially, we are specifying the behavior of our program. The total credit a customer receives is the sum of all credit transactions for that customer. This suggests creating a definition which will take a customer and the list of transactions and return the total amount of credit for that customer. Such a definition would then be used in the assertion we wish to make about the customer's balance. The definition could take the following form:

```
Define Total_Credit(c:Customer, TList:Seq Of Transaction) =
        If Size(TList) = 0 Then 0
        OrIf Head(TList).customer_number = c.customer_number And
                Type(Head(TList)) Same transactions_in:credit Then
                        Head(TList).amount + Total_Credit(c,Tail(TList))
        Else Total_Credit(c,Tail(TList))
        Fi;
```

The keyword *Define* introduces a definition. The definition takes two arguments. The first is of type Customer and the second is a sequence of Transaction. If the size of the transaction sequence is 0, then there are no transactions. The definition then returns 0. Otherwise, if the transaction at the head of the sequence has the same customer number as the customer, meaning that this transaction deals with this customer, and the transaction is of type credit (the *Type* function returns the actual type of a class object) we add the amount of this transaction to the total credit of the remaining transactions (the *Tail* function returns the sequence which is the argument without its head). If the transaction does not concern this particular customer or the transaction is not a credit, then we return the total credit of the remaining transactions only.

The definition of Total_Credit is recursive. It is guaranteed to terminate since the definition is applied to a smaller collection each time it is called (the *Tail* of a sequence is always smaller than the sequence itself, if the sequence is non-empty), and the definition does not call itself when an empty sequence (of size 0) is encountered.

The operator *Same* was used in the Total_Credit definition. When comparing two values, the operator "=" is used. When comparing two collections, the operator *Same* is used. The function *Type* returns the collection of all objects in the structure instance which are of the same type as the function argument. A class or node name returns the collection of all objects in the structure instance which are of the same type as the class or node. Thus in the definition of Total_Credit, *Type*(*Head* (TList)) and credit both return collections of objects and must be compared with *Same* rather than "=".

With this definition in hand, we can now make the assertion that the balance of a customer at output is the balance the customer had at input plus any credit it received.

```
-- All customers are properly credited.
Assert ForAll c_out In customers_out:Customer Do
        ForAll c_in In customers_in:Customer Do
                If c_in.customer_number = c_out.customer_number Then
                        c_out.balance = c_in.balance + Total_Credit(c_in, transactions_in:Root.list)
                Else True Fi
        Od Od;
```

The meaning of the above assertion should be clear except for two points. First, note the *Else True*. Every *If* clause must have an *Else* clause attached. In this case, if the customer numbers are not equal, then the program is fine. So we assert *True* in the *Else* clause. There may be situations where it makes sense to assert *False*. Second, the reserved word *Root* was used. *Root* refers to the root object of the specified structure. If no structure is specified, it refers to the root object of the structure in which the assertion or definition appears. In this case, the port prefix transactions_in specifies we mean the root of the structure associated with the port transactions_in, which is the transactions structure. The root of this structure is of type transaction_list. This type has an attribute of type *Seq Of* Transaction. That is why the *Root*.list is specified as an argument. This dotted expression has type *Seq Of* Transaction since the specified root has an attribute called list which has this type.

In the Total_Credit definition, *OrIf* was used. The *OrIf* form is a shorthand which is recommended. Every use of *If* must be ended with a *Fi*. Use of *OrIf* does not require a *Fi*. Thus *OrIf* is preferable to *Else If*.

As another example of the use of definitions, we can use a definition to help us assert that the number of bills generated equals the number of debit transactions. We can create a definition which will return the number of debit transactions.

```
Define Num_debits(TList:  Seq Of Transaction) =
        If Size(TList) = 0 Then 0
        OrIf Type(Head(TList)) Same transactions_in:debit Then 1 + Num_debits(Tail(TList))
        Else Num_debits(Tail(TList)) Fi;
```

With this definition, we can now assert:

```
Assert Size(bills_out:Root.list) = Num_debits(transactions_in:Root.list);
```

Assertions may often be written in several ways. For instance, the above assertion could have been written without using the Num_debits definition:

```
Assert Size(bills_out:Root.list) = Size(transactions_in:debit);
```

Collection-returning Definitions Definitions may also return *collections* of objects. A collection is similar to an IDL set, differing in three important ways. First, an attribute may have an IDL set as a value, but may not have a collection as a value. Secondly, an IDL set is declared explicitly using "*Set Of*"; a collection is determined implicitly by the IDL translator. Finally, collections exist only within assertions.

One may define a collection of objects and then make some assertion about the objects in the collection. For example, we wish to assert that a bill is generated for each debit transaction. First we can define a definition which returns the collection of all objects which are debit transactions. Then we can assert that there exists a bill for each of these transactions.

```
Define Debits(TList:  Seq Of Transaction) =
        If Size(TList) = 0 Then Empty
        OrIf Type(Head(TList)) Same transactions_in:debit Then
                Head(TList) Union Debits(Tail(TList))
        Else Debits(Tail(TList)) Fi;
```

The usual set operations (including *Union*, used above) are available to use with collections of objects which exist explicitly in the IDL structure specification or which are defined in the assertion language. If the transaction sequence is empty, the definition returns *Empty*. This denotes the empty collection, or the collection of no objects. Returning 0 would not make sense in this context, since the definition is returning a collection of objects.

In fact, there is a cleaner, non-recursive way of stating this (*Members* applied to sequences is a UNC extension):

```
Define Debits(TList:  Seq Of Transaction) = Members(TList) Intersect debit;
```

We may now make our assertion:

```
-- A bill is generated for every debit transaction.
Assert ForAll deb In Debits(transactions_in:Root.list) Do
        Exists b In Members(bills_out:Root.list) Do
                b.billee.customer_number = deb.customer_number And b.amount = deb.amount
        Od Od;
```

The above assertion makes use of the collection returning definition, and the two level deep dotted expression. b.billee is of type Customer. Thus, b.billee has an attribute called customer_number, and the reference to b.billee.customer_number is permitted. *Members* is a supplied function which takes a set or sequence (as in this case) and produces the collection containing all objects in that set or sequence. Use *Members* whenever you are interested in the objects contained in a set or sequence, rather than the set or sequence as a single object.

With the Debits definition above, there is yet another way to assert that that the number of bills equals the number of debit transactions:

```
Assert Size(bills_out:Root.list) = Size(Debits(transactions_in:Root.list));
```

2.2 Translating the Specification

The second step in using the IDL system is to translate the collection of data and process specifications with the IDL translator into a set of data type declarations in the target programming language that expresses the same functionality and to generate a set of run-time support routines tailored for manipulating target-language versions of the user's data structures. Figure 12 shows this portion of the whole process of using the IDL system.

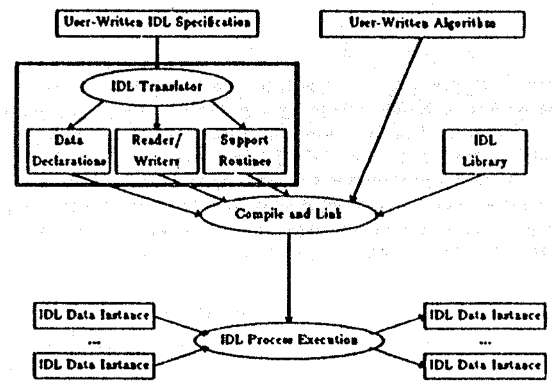


Figure 12 The Specification Translation Step

2.2.5 Using the IDL Translator

To use the IDL translator, the user enters a command line of the following form.

idlc [*options*] *file1* [*file2* ...]

The options portion of the command specifies zero or more options to change the default flow of the translator. The list of one or more files contains the IDL specification (described below) of data structure and process specifications to be processed as a unit.

By default, the IDL translator parses the input file or files making up the IDL specification for syntactic and semantic correctness. For each process specification encountered, the IDL translator (with C as the default target programming language) creates two files; a "(process name).h" file containing the declarations of the data types and a "(process name).c" file containing the definitions of process-specific code for the run-time support routines. At the end of the generation of the ".h" and ".c" files, the IDL translator compiles each ".c" file with the C compiler into a ".o" file and deletes the ".c" files. Several options exist for changing the default behavior.

As an example, to process the specification shown in Figure 10, the user enters the command shown below (characters typed by the user are shown in **bold**).

idlc -v **billing.idl**

In response to this command, the IDL translator creates the billing.h and billing.c files and then compile the billing.c into a billings.o file. After the compliation, the billing.c is deleted. The billing.h file contains declarations for all data types and constants. The billing.o file contains the code for the ports for the billing process. The -v option requests messages indicating what the IDL translator is doing.

An *IDL specification* is a collection of structure and process specifications that the user processes as a unit with the IDL translator. The specification resides in one or more files. The user should process all of the files making up the IDL specification as one group.

The collection may contain the specifications for many different structures and processes at the same time. These structures and processes may make up a complete system of cooperating processes or only a subset of such a system. Alternatively, the collection may contain only structure specifications. In this case the only action that the IDL translator will perform is to check the syntactic and semantic correctness of the specifications. Similarly, if additional structures are included in the collection that are not referred to by any process, they will be checked by the IDL translator for syntactic and semantic correctness. Of course, the declarations for all structures mentioned in any of the process declarations must be included in the IDL specification being processed and the names of all distinct structures and processes within the collection must differ from one another.

2.2.6 *Declarations Produced by the IDL Translator*

The user programs his algorithm in terms of the data type declarations produced by the IDL translator. While programming, he makes use of the tailored set of run-time support routines to allocate new instances of data structures, to maintain sets and sequences composed of them, and to read and write them from external storage. This section examines the declarations for the C language [Kernighan & Ritchie 1978] produced by the IDL translator from the specification shown in Figure 10, and provides an overview of the runtime support routines. The declarations are extracted from the file billing.h. The organization will follow that of Section 2.2. A reference manual for the C interface to IDL discusses these aspects in greater detail [Shannon, et al. 1985]. Other target languages are also available; each is described in an associated reference manual.

The "⟨process⟩.h" file produced by the IDL translator is usually quite long and repetitious. However, with a little experience the user need not look at the contents of the file to use the declarations contained in it. Each IDL construct is mapped into a C declaration in a consistent fashion: all names follow the convention of a capital letter followed by the identifier. An appendix in the reference manual lists the meaning of all the single letter prefixes.

IDL constructs are mapped to a combination of C declarations and macro and constant declarations. Nodes are mapped to C *structs*, and their attributes are mapped to members of the *struct*. The basic types of *Integer* and *Rational* are mapped to the C types of *int* and *float*. *Boolean* is mapped to *int*, taking on the values 0 (*False*) and 1 (*True*). *String* is mapped to *String*, which is a new type effectively encapsulating a pointer to a null-terminated character string. Finally, an initialization macro and a manifest constant specifying the type are defined. The following is generated for the node state_customer (c.f. Figure 5).

```
typedef struct Rstate_customer * state_customer;
# define Kstate_customer 16
struct Rstate_customer {IDLnodeHeader IDLhidden;
      String name;
      String address;
      int customer_number;
      Boolean active;
      float balance;
      int state_code;
};
# define Nstate_customer ...
```

Node references (state_customer) are represented as pointers to the appropriate struct (Rstate_customer). A second representation of unattributed nodes, as integers, is beyond the scope of this tutorial.

An instance of a structure may form a graph by having nodes reference other nodes through attribute values. The structures defined in Figure 10 are rather simple, in that the only node references are in lists. However, sharing is still possible. For instance, a Customer referenced in the billee attribute of a bill may be one of the Customers found on the list referenced by the list attribute of a customer_list (in fact, the algorithm discussed in Section 2.4 ensures that Customer nodes are shared in this way.) It is even possible in the general case to have cycles, where one node indirectly references itself through one or more attributes (note however that instances of the structures in Figure 10 cannot have cycles). The values of the basic types are always shared; conceptually there is only one instance of each value.

For attributed nodes, four identifiers are declared: the struct Rstate_customer, the pointer state_customer, the manifest constant Kstate_customer, and the initialization macro Nstate_customer. The IDLhidden field in the struct is used by the runtime routines, and should be ignored by the user. The manifest constant is returned by the typeof function, when given a particular node of that type. The initialization macro returns a reference to the node. The names of these constants, structs, pointers, and macros follow the convention described earlier.

Sets and sequences are represented as linked lists or arrays; only the linked list representation will be discussed here. The type Seq Of bill is mapped into

```
typedef struct IDLtag4{
      struct IDLtag4 *next;
      bill value;
} Cbill, *Lbill;
# define SEQbill Lbill
# define inSEQbill(billseq,billvalue) ...
# define initializeSEQbill(billseq) ...
# define appendfrontSEQbill(billseq,billvalue) ...
# define appendrearSEQbill(billseq,billvalue) ...
# define orderedinsertSEQbill(billseq,billvalue,billcompfn) ...
...
```

Over a dozen macros specific to the sequence are defined for each sequence; sets are accompanied by similar macros. Each macro name is structured as a verb followed by the characters "SEQ" followed by the IDL identifier, in this case "bill". The initializeSEQbill macro must be called before an attribute or variable of type *Seq Of* bill is accessed. The attribute or variable is the one argument of the macro. The appendrearSEQbill macro takes two arguments: an attribute or variable of type *Seq Of* bill, and an expression of type bill. The bill is added to the end of the sequence. Finally, the foreachinSEQbill macro is an iterator in the fashion of a for statement. It takes three arguments: a *Seq Of* bill to iterate over, a variable of type *Seq Of* bill used to keep track of the iteration, and a variable of type bill, which will be successively assigned a bill in the sequence. It is used in this way:

```
SEQbill    tempSEQbill;
bill  abill;
...
foreachinSEQbill(abill_list->list, tempSEQbill, abill) {
/* do something with abill */
}
```

If the sequence abill_list->list contained five nodes, then "do something" would be executed five times, each with a different node referenced by abill.

Classes are mapped into C unions. The class Customer in Figure 6 is mapped into the following by the IDL translator:

```
typedef struct HCustomer * HCustomer;
typedef union {
      int IDLinternal;
      HCustomer IDLclassCommon;
      commercial_customer Vcommercial_customer;
      Government_customer VGovernment_customer;
      state_customer Vstate_customer;
      federal_customer Vfederal_customer;
} Customer;

struct HCustomer {
      IDLnodeHeader IDLhidden;
      String name;
      String address;
      int customer_number;
      Boolean active;
      float balance;
      };
```

The IDLinternal member and IDLhidden field should be ignored by the user. The IDLclassCommon member contains those attributes common to the class, allowing the following:

```
Customer thisCustomer;
thisCustomer.IDLclassCommon->active = True;
```

Note that in Figure 10, five attributes were associated with Customer; those are the attributes available through IDLclassCommon. Had Customer inherited any attributes from other classes, these too would have been available through IDLclassCommon. The five attributes are available for variables of type Government_customer for this reason, even though they aren't associated with Government_customer directly.

Attributes associated with subclasses are *not* directly available to variables of a class. Hence, the attribute state_code cannot be directly accessed through thisCustomer. The remaining members of the union are used when accessing attributes not common to the class or when passing a class as a parameter to a function. To access the state_code attribute, use

```
if (typeof(thisCustomer)==Kstate_customer){
      thisCustomer.Vstate_customer->state_code ...
}
```

The if statement uses the typeof function to determine that the Customer referenced by the variable thisCustomer was indeed a state_customer node. Such a test is necessary because no type-checking is done when the Vstate_customer member is selected. Omitting the test may result in incorrect code.

The union members are also useful when calling routines expecting subclasses or superclasses of a given class. If a subclass (e.g., state_customer) is expected, a type check is required. If a superclass is expected (an example is calling a routine expecting a Government_customer, passing a federal_customer as a parameter), the type check is not required.

To pass the variable thisCustomer to the routine

```
ProcessStateCustomer(aCustomer)
state_customer aCustomer;
```

use

```
if (typeof(thisCustomer)==Kstate_customer)
      ProcessStateCustomer(thisCustomer.Vstate_customer);
```

To call the routine

```
ProcessGovernment_customer(aGC)
Government_customer aGC;
```

with a variable of type federal_customer, only a cast is needed:

```
federal_customer afc;
...
ProcessGovernment_customer((Government_customer)afc);
```

The user must keep the class membership tree (c.f., Figure 6) in mind when doing these type conversions. This is a somewhat tedious but not troublesome task.

Each IDL process is mapped to a C program. The structures used by the process are mapped to struct and union declarations for that program. Ports are mapped to C functions. Input ports (*Pre*) are mapped to functions taking a file pointer and returning a node of the root type for the structure. Output ports (*Post*) are mapped to functions taking a file pointer and a node for the root fo the structure instance. The four ports declared in Figure 9 would be mapped to these declarations:

```
void bills_out();
void customers_out();
transaction_list transactions_in();
customer_list customers_in();
```

2.3 Writing the Algorithm

Writing the algorithm for the process in the target programming language is the next step. Figure 13 illustrates this portion of the process of using the IDL system. The flow of the billing program is illustrated in Figure 14.

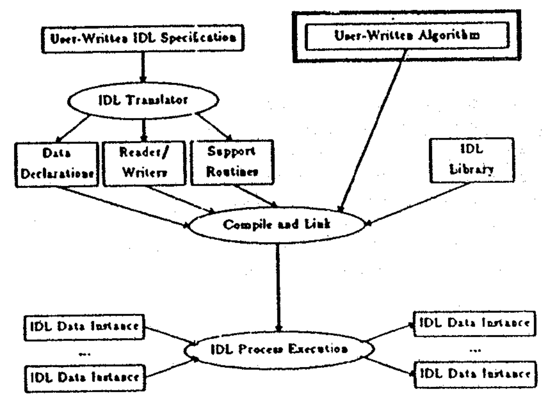


Figure 13 The Algorithm Writing Step

The algorithm in this example is a simple one. It reads in a customer list and a sequence of transactions, through the customers_in and transactions_in ports, respectively. Then it steps through the transactions, either crediting the customer or generating a bill, according to the type of the transaction. Finally, it writes out the updated customer list and the list of bills, through the customers_out and bills_out ports, respectively. The algorithm as coded in C is given in Figure 15.

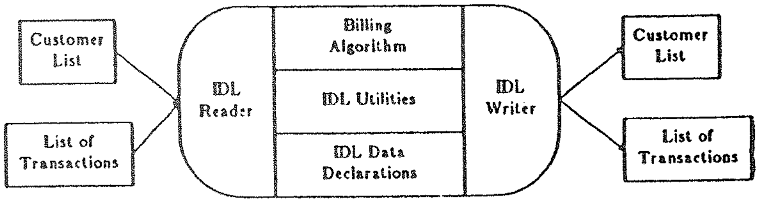


Figure 14 Conceptual View of the Billing Process

```
/* file billing/algorithm.c */
#include <stdio.h>
#include "billing.h"

main ()
{
      customer_list                    thisCL;
      transaction_list                 thisTL;
      bill_list                        thisBL;
      Customer                         thisCustomer;
      Transaction                      thisTransaction;
      bill                             thisbill;
      SEQCustomer                      remainingCustomers;
      SEQTransaction                   remainingTransactions;
      FILE *c_in, *t_in, *c_out, *b_out;

/* input the initial customer list */
      c_in = fopen("customers.in", "r");
      thisCL = customers_in(c_in);
/* input the transactions list */
      t_in = fopen("transactions.in", "r");
      thisTL = transactions_in(t_in);

/* initialize bills */
      thisBL = Nbill_list;
      initializeSEQbill(thisBL->list);

/* process the transactions */
      foreachinSEQTransaction(thisTL->list, remainingTransactions,
                                        thisTransaction) {
              /* find the Customer mentioned in the transaction */
          foreachinSEQCustomer(thisCL->list, remainingCustomers, thisCustomer)
              if (thisCustomer.IDLclassCommon->customer_number
                      == thisTransaction.IDLclassCommon->customer_number) break;

          switch (typeof(thisTransaction)) {
          case Kcredit:                 /* credit the Customer */
                  thisCustomer.IDLclassCommon->balance
                          += thisTransaction.IDLclassCommon->amount;
                  break;
          case Kdebit:                  /* generate a bill */
                  thisbill = Nbill;
                  thisbill->billee = thisCustomer;
                  thisbill->amount = thisTransaction.IDLclassCommon->amount;
                  appendrearSEQbill(thisBL->list, thisbill);
                                break;
                    /* can't be anything else */
                    }
          }

/* write out the updated customer list */
      c_out = fopen("customers.out", "w");
      customers_out(c_out, thisCL);
/* write out the list of bills */
      b_out = fopen("bills.out", "w");
      bills_out(b_out, thisBL);
      exit(0);
}
```

Figure 15 The C Algorithm for the Billing Process

The comments explain what is going on, so we will only make a few observations here. Since a `bill_list` is being created, it must be allocated (`Nbill_list`) and the sequence attribute initialized (`initializeSEQbill`). The outer `foreachinSEQTransaction` successively assigns transactions to `thisTransaction`; the inner `foreachinSEQCustomer` successively assings customers to `thisCustomer`, attempting to find the customer with the correct `customer_number` (it is assumed to exist). To generate a bill, one is first allocated, then its attributes are filled in, then it is appended to the list of bills. Finally, both output ports are called. The declarations, constants, and macro definitions produced by the IDL translator are used extensively. Because of the consistent naming, a programmer experienced with the IDL system would need only the IDL specification to code the algorithm.

2.4 Compiling the Process

Compiling and linking the the code for his algorithm along with the IDL-produced data declarations and run-time support routines to produce an executable program is the fourth step.

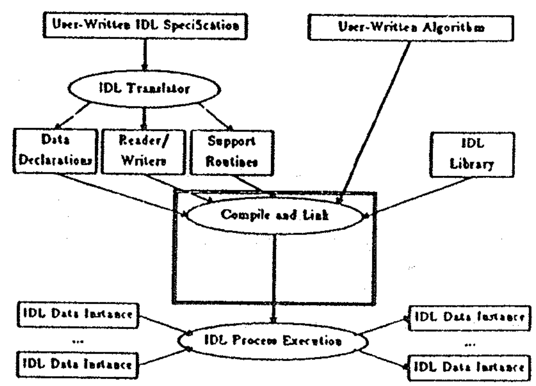


Figure 16 The Process Compilation Step

In the compilation stage, the user compiles his code and links it with the code produced by the IDL translator and with a library of generic IDL system routines. For example, a command to compile and link an application built on the specification of Figure 10 would be the following:

```
cc algorithm.c billing.o /usr/softlablib/libidl.a -o billing
```

In this example, the code for the user's algorithm is assumed to reside in a file named `algorithm.c`, the IDL-generated routines reside in the file named `billing.o`, and the library of generic IDL system routines resides in the `libidl.a` file. The output of this command will be an executable command in the file named `billing`. The pathname for the library file will vary from system to system.

2.5 Running the Process

The fifth and last step in using the IDL system is to run the program to process instances of the data structures represented externally in the ASCII External Representation Language.

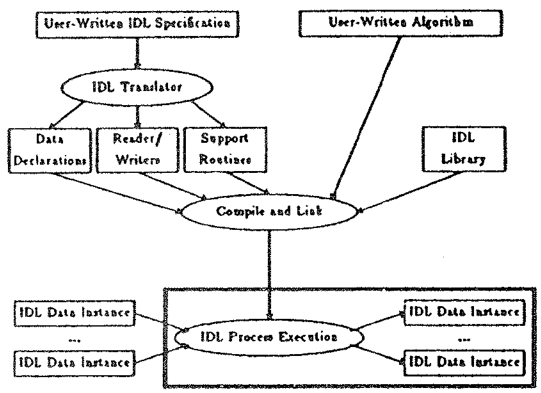


Figure 17 The Process Execution Step

Before this step can be performed, instances of the input structures must be available. The billing process referenced two input structures: `customers` and `transactions`. Normally, instances of input structures are written by other processes. In this example, they have been manually constructed.

The IDL ASCII External Representation Language is the standard representation scheme for external instances of IDL data structures. Instances are actual data values whose structure corresponds to the specified data structure. The external representation of an instance of a data structure consists of a list of each node in the structure. Each of the nodes in the list are given a unique temporary label. The first node in the list must be a reference to the root node. The order of subsequent nodes in the list is not significant. The "#" sign in the first column of a line signals the end of nodes forming the structure. Hence, more than one instance of one or more structures may reside in the same file. In a valid structure instance, all nodes referred to must have a definition.

A node is represented as a name followed by a list of attribute-reference pairs. The name of a node is the type name of that node. Each attribute is explicitly named and the name is followed by a reference to the corresponding attribute value. The name of an attribute is the name specified in the structure declaration.

The value of an attribute of IDL basic type is the external representation of that value. Special care was taken to ensure that the representation of *Rational* and *String* values are machine independent. The value of an attribute of node type is an indirect reference to that node. A reference to a node consists of the label of that node followed by a "^" character. Forward references are permitted. A list of types or nodes bracketed with braces "{}" represents a set-valued attribute. A list of types bracketed with angle brackets "<>" represents a sequence-valued attribute. Comments are preceded with "-- ".

An example of an instance of the `customers` structure cast in the ASCII External Representation Language is shown below

```
L1:  customer_list [list < L2^L3^L4^>]
L2:  commercial_customer [
name "Innovation, Inc.";
address "Freedom Trail";
active TRUE;
customer_number 1;
balance 9546782.00;
industry_code 12
]
L3:  state_customer [
name "Department of Obfuscation";
address "Bereaucracy Boulevard";
active FALSE
customer_number 2;
balance -1000000000.86;
state_code 50
]
L4:  federal_customer [
name "Office of the Director, OMB";
address "Wonderland";
active FALSE
customer_number 3;
balance -1000000000000.13;
agency_code 1348903
]
#
```

In this example a list of three customers is shown. The name of the first customer is "Innovation, Inc.", the second, "Department of Obfuscation", and the last, "Office of the Director, OMB". Note that only the nodes in this instance are represented externally. Classes for the internal representation can be reconstructed by the reader generated by the IDL translator solely from its knowledge of the structure specification.

We also need an instance of the `transactions` structure:

```
L1:  transaction_list [list <L2^L3^>]
L2:  credit [
customer_number 2;
date 31285;
amount 22354.44;
tax_status {}
]
L3:  debit [
customer_number 1;
date 31485;
amount 332.12;
tax_status {state_sales_tax}
]
#
```

This instance contains two transactions, a `credit` to customer number 2 and a `debit` from customer number 1.

After we execute the billing process

billing

we note that two files have been created: customers.out and bill.out. The first contains

```
customer_list[list <commercial_customer[name "Innovation.Inc.";
address   "Freedom Trail";
customer_number   1;
active TRUE;
balance   9.54676E+06;
industry_code 12]
state_customer[name "Department of Obfuscation";
address   "Bureaucracy Boulevard";
customer_number   2;
active FALSE
balance   1.1E+10;
state_code 50]
     federal_customer[name :Office of the Director, OMB";
address   "Wonderland";
customer_number   3;
active FALSE
balance   1.1E+13;
agency_code 1348903]
>]
#
```

and the second

```
bill_list[list  <bill[billee commercial_customer[name"Innovation, Inc.";
address   "Freedom Trail";
customer_number   1;
industry_code 12]
;
amount 332.12]
>]
#
```

That this output is correct is left as an exercise to the reader.

3 Advanced Features of IDL

IDL has several advanced facilities that have not yet been introduced. Four of those facilities will be briefly described in this chapter: one for extending the standard set of attribute types with user-implemented private types, another for deriving and refining new data structures from one or more previously defined data structures, a third for creating user-supplied process invariant structures, and a fourth for naming assertions. As was shown in the last chapter, the IDL user need not employ these facilities in solving problems with IDL; however, their use can lead to more precise and powerful specifications.

3.1 Private Types

The private type facility allows the user to extend the standard set of attribute types provided by IDL. The standard set of attribute types provided by IDL are the basic types (*Boolean*, *Integer*, *Rational*, and *String*) and the structured types (*Set Of* and *Seq Of*). With private types, the user may augment this standard set to include more specialized types.

The user first declares a name to identify the private type. He then specifies an external representation for the private type in terms of both the standard set of attribute types provided by IDL and in terms of node and class types expressed wholly in terms of the standard set of attribute types. Finally, he specifies the name of a file containing the target programming language data declarations that define the internal representation of his type and provides appropriately-named routines to map the external representation for the private type to and from the internal representation.

An example of the use of a private type is shown in Figure 18. In this example, the declaration of the customers data structure of Figure 10 is modified to make use of a date internal representation for the date attribute. The external representation of the date is kept as an integer.

```
Structure transactions Root transaction_list Is
    . . .
    Type date_type;
    Transaction =>  customer_number :  Integer,
                    date            :  date_type,
    . . .
    For date_type Use External Integer;
    For date_type Use Package DatePackage;
    For date_type Use Type packeddate;

End
```

Figure 18 A Private Type Declaration

In this example, the declaration of the private type, date_type, is introduced by the IDL keyword *Type*. The next statement declares that the date_type private type will be represented externally as the IDL type *Integer*. The third statement declares that the target programming language data type declarations for the internal representation of the date_type private type will be found in a file named DatePackage.h (the ".h" file extension is generated by the translator). The user would be responsible for linking the appropriately named routines for mapping the *Integer* external representation to and from the packeddate internal representation for input and output of instances of the data structure [Shannon, et al. 1985].

3.2 Derivation and Refinement of Structures

The derivation and refinement facilities allow the user to declare a new data structure in terms of one or more previously declared data structures. This makes the relationships between different data structures clearer. It also allows the user to avoid multiple copies of the same information and the attendant problems in keeping all copies consistent with one another. The user can record a set of declarations in just one place and use derivation and refinement to make the declarations of related data structures.

Derivation allows the user to copy the node and class declarations from several structures and then add or delete attributes to node types, members to class types, whole node types, or whole class types. An example of the usefulness of derivation would be deriving the bills data structure of Figure 10 from the customers data structure. The declaration for the derivation of the bills structure from the customers structure is shown in Figure 19.

In this example, the derivation of the bills data structure from the customers data structure is signaled by the IDL keyword *From*. The three statements beginning with the IDL keyword *Without* delete parts of the customers data structure. The first *Without* statement deletes the customer_list node from the new data structure entirely. The next two delete the active and balance attributes of the node members of the Customer class. The two node productions add the bill_list and bill nodes to the new data structure. The result of this declaration of the bills data structure is exactly the same as that in Figure 10. Note though how much more clearly the relationship between the customers and bills data structures is shown by this declaration than that originally given in Figure 10.

```
-- Specification for the bills structure.

Structure bills Root bill_list From customers Is

    Without customer_list;
    Without Customer => active;
    Without Customer => balance;

    bill_list       => list   :  Seq Of bill;

    bill            => billee :  Customer,
                       amount :  Rational;

End
```

Figure 19 An Example of Structure Derivation

Refinement allows the user to copy the node and class type declarations of several structures and add (but not delete) new attributes to node types, new members to class types, or entirely new node or class types. An example of the usefulness of refinement is given below.

3.3 User-supplied Process Invariant Structures

The *invariant* data structure of a process is a union of the declarations of all structures referred to in that process. The invariant is a union in the sense that for every node and class type declared in any of the other structures referred to in the process, a declaration for that node or class type exists in the invariant data structure. Furthermore, the set of attributes declared for a node type in the invariant data structure is at least the union of the attributes declared for that node type in all of the other other structures referred to in the process. Likewise, the set of direct class members for a class type in the invariant data structure is at least the union of the direct class members declared for that class type in all of the other structures referred to in the process.

The purpose of the invariant data structure is to simplify the automatic generation of routines to manipulate instances of the node and class types for use within an instance of an IDL process and to simplify the user's access to the data values within instances of the node and class types. Within an IDL process, only one implementation of a node or class type exists. The declaration of the node or class type that is actually implemented is that appearing in the invariant data structure no matter how many different declarations are given in the port data structures referred to by the process.

When writing the target programming language code for an instance of a process, the user logically views his data as being organized into instances of the one or more data structures referred to within the process. In this view, an instance of a particular node or class type name that forms part of an instance of a particular data structure has only those attributes or members that are declared within the specification for that data structure.

This has several advantages. First, the user need not qualify his node and class types with the particular structure that he wishes to view them in. Second, IDL need only provide one set of run-time support routines for manipulating a node of a certain type. Third, the user may detach an instance of a node of a certain type from an instance of one structure type and reattach it to an instance of another structure type.

A disadvantage is that the space required for the invariant's version of a node or class type may be much larger than the minimum that would be required for the version of the node or class type given in a particular non-invariant data structure.

IDL enforces the logical view of the data structures in the input and output of instances of the data structures. In the example of Figure 10, instances of the state_customer node would be read in as part of an instance of the bills data structure with the expectation that they contained values only for the name, address, customer_number, and state_code attributes because the bills data structure declares that the Customer node only has these attributes.

The user need not worry about generating the invariant structure for a process. Unless he takes special actions as described below, the IDL translator will automatically generate the invariant.

A user may supply his own process invariant structures rather than using the invariant automatically generated by the IDL translator. This facility allows the user to define new nodes and classes that do not appear in any of the other structures referred to by the process or to add attributes and members to nodes and classes already defined in those structures. These new nodes and classes or attributes and members can be used for intermediate calculations within the process without cluttering the external definition of the structures read and written by the process.

The user may derive the invariant data structure from the set of data structures referred to by the process according to the following sequence of steps:

1. The user gives the new invariant data structure a unique name.

2. For every node or class type for which there is a declaration in only one of the structures used by this process, the user copies that declaration into the specification for the new invariant structure.

3. For every node for which there is a declaration in more than one of the structures used by this process, the user writes a declaration for the node type in the new invariant data structure such that the new declaration has all of the attributes found in any one or more of the declarations in the different structures. If two structures have a declaration for the same-named node type and if both declarations have attributes of the same name but these same-named attributes of the same-named node types have different types, then there is an irresolvable conflict. The user must either adjust the types of all same-named attributes in same-named nodes in all structures used by the process to be of the same type, or else change the names of the offending attributes or of the offending nodes such that this is no longer true.

4. The user adds attributes to the existing nodes and or members to the existing classes or adds new nodes and classes consistent with the rules given for structures above. The properties and usefulness of these added attributes and members will be explained below under the discussion on the purpose of the invariant.

5. The user designates a node or class type as the root of the invariant data structure. As for any other structure, all node and class types must be reachable from the root node or class type. It may be necessary for the user to create a special node or class type to serve as the root.

6. The user checks that the class memberships for the new invariant structure form a forest of non-intersecting trees just as for any other structure. If this is not true, then the user must either change the class structures in the individual port structures and in the invariant structure or change the names of the offending classes in the individual port structures and in the invariant structure in such a way that this will be true.

Alternatively, if any data structure referred to in a process meets all of the criteria given above for the invariant data structure, then that structure may be designated by the user as the invariant data structure for the process.

An example of an invariant data structure for the billing process of Figure 10 is shown in Figure 20. This invariant has been formed from the customers, transactions, and bills data structures. Note the creation of the master_node node type to serve as the root of the invariant. Also note the addition of the partial_total attribute to the Customer node type. This attribute is not declared in any of the non-invariant data structures referred to by the process.

In this example, the refinement of the billing_inv data structure from the bills, customers, and transac-

```
-- Specification for the billing invariant structure.

Structure billing_inv Root master_node Refines customers transactions bills Is

    master_node => customers     : customer_list,
                   transactions  : transaction_list,
                   bills         : bill_list;

    Customer    => partial_total : Rational;

End

Process billing Inv billing_inv Is
    . . .
End
```

Figure 20 A User-Supplied Invariant for the billing Process

tions data structures is signaled by the IDL keyword **Refines**. The first node declaration statement adds the master_node node to the new data structure. The second node declaration statement adds the partial_total attribute to all node type members of the Customer class.

3.4 Naming Assertions

It is possible to name assertions. For example:

```
Range_check Assert ForAll CF In federal_customer Do CF.agency_code <= 100 Od;
```

Range_check is the name of the assertion. The naming of assertions is necessary if one wants to make use of the Without statement in an IDL specification. For instance, if the output of a process has the same structure as the input, but certain assertions should no longer hold in the output, one could state in the output specification: Without Range_check;

This assertion would then not be checked in the output structure, though all other assertions would be. In addition, naming assertions also facilitates communication between people.

3.5 Summary

The above has been a brief introduction to a few of the more advanced facilities of IDL. For a complete description of these and other IDL facilities, see "The IDL Formal Definition" [Nestor, et al. 1982]. In addition to these target programming language independent facilities, there are several target programming language-specific advanced facilities that the experienced user will find useful in constructing his applications.

4 Acknowledgements

Much of the material on specifying data structures in IDL is similar to portions of the IDL formal definition [Nestor, et al. 1982] and Lamb's dissertation [Lamb 1983]. In particular, Figures 2 and 3 are modified versions of figures that originally appeared in these documents, and are used with permission. Ilsoo Ahn, Ed McKenzie, John Nestor, Karen Shannon and especially David Lamb provided comments on previous versions. Finally, Leigh Pittman and Pamela Manning helped with the formatting.

5 Bibliography

[Birtwistle, et al. 1973] Birtwistle, G.M., O.J. Dahl, Myhrhaus B. and K. Nygaard. *Simula Begin*. Philadelphia, PA: Averbach Publishers, Inc., 1973.

[Brosgol et al. 1980] Brosgol, B.M., J.M. Newcomer, D.A. Levine Lamb, D., M.S. Van Deusen and W.A. Wulf. *TCOL Ada: Revised Report on An Intermediate Representation for the Preliminary Ada Language*. Technical Report CMU-CS-80-105. CMU. Feb. 1980.

[Butler 1983] Butler, K.J. *DIANA Past, Present, and Future*, in *Lecture Notes in Computer Science Ada Software Tools Interfaces*, Ed. G. Goos and J. Hartmanis. Workshop, Bath: Springer-Verlag, 1983, pp. 3-22.

[Cattell et al. 1980] Cattell, R., D. Dill, P. Hilfinger, S. Hobbs, B. Leverett, J. Newcomer, A. Reiner, B. Schatz and W. Wulf. *PQCC Implementor's Handbook*. Carnegie-Mellon University, 1980.

[Goldberg & Robson 1983] Goldberg, A. and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Goos & Wulf 1981] Goos, G., W.A. Wulf and (eds). *Diana Reference Manual*. Technical Report CMU-CS-81-101. Computer Science Department, Carnegie-Mellon University. Mar. 1981.

[Kernighan & Ritchie 1978] Kernighan, B.W. and D.M. Ritchie. *The C Programming Language*. Englewood Cliffs, NJ 07632: Prentice-Hall, Inc, 1978.

[Lamb 1983] Lamb, D.A. *Sharing Intermediate Representations: The Interface Description Language*. PhD. Diss. Computer Science Department, Carnegie-Mellon University, May 1983.

[Leverett et al. 1980] Leverett, B., R.G.G. Cattell, S.O. Hobbs, J.M. Newcomer, A.H. Reiner, B.R. Schatz and W.A. Wulf. *An Overview of the Production Quality Compiler-Compiler Project*. *Computermag*, 13, No. 8, Aug. 1980, pp. 38-49.

[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.

[Persch et. al. 1980] Persch, G., M. Winterstein, M. Dausmann, S. Drossopoulou and G. Goos. *AIDA Reference Manual*. Technical Report Nr. 39/80. Universitaet Karlsruhe. Nov. 1980.

[Shannon, et al. 1985] Shannon, K., T. Maroney and R. Snodgrass. *Using IDL with C*. SoftLab Document No. 6. Computer Science Department, University of North Carolina at Chapel Hill. May 1985.

[Snodgrass 1985] Snodgrass, R., editor *IDL Manual Entries (Version 2.0)*. SoftLab Document No. 15. Computer Science Department, University of North Carolina at Chapel Hill. Dec. 1985.

[Stroustrup 1986] Stroustrup, B. *The C++ Programming Language*. Reading, MA: Addison-Wesley Pub. Co., 1986.

[Wulf et al. 1971] Wulf, W.A., D. B. Russell and A. N. Habermann. *Bliss: A Language for System Programming*. *Communications of the Association of Computing Machinery*, 14, Dec. 1971, pp. 780-790.

[Zorn 1985] Zorn, B. *Experiences with Ada Code Generation*. Technical Report UCB/CSD 85/249. University of California, Berkeley. June 1985.