Mapping the Interface Description Language Type Model into C

KAREN SHANNON AND RICHARD SNODGRASS, SENIOR MEMBER, IEEE

Abstract—**The Interface Description Language (IDL) is a notation for describing the characteristics of data structures passed among collections of cooperating processes in a programming environment. In this paper we discuss a mapping from IDL to C data structures and macro definitions that supports the full language and is type safe and runtime efficient, but is not particularly compile-time efficient nor easy to use. We then propose that the mapping be performed in a new preprocessor, thereby achieving all five goals.**

Index Terms—**Interface Description Language (IDL), intermediate representation, programming environment, type model.**

I. Introduction

The Interface Description Language (IDL) is a notation for describing the characteristics of data structures passed among a collection of cooperating processes, such as the phases of a compiler [15], [16], [19], [29]. 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. The best known IDL specification is Diana, the defacto standard intermediate representation of Ada programs [8]. Diana has been used in most Ada compilers, including those implemented at Bell Labs, Burroughs, the University of California at Berkeley, Intermetrics, the University of Karlsruhe, Rolm, SofTech, and Verdix. 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 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 user writes programs in terms of the target language data declarations and utilities produced by the IDL translator. These programs process instances of the IDL-specified data structures residing on external storage. IDL has been used to specify intermediate representations communicated between phases of various compilers, and has found application in other tools (e.g., a cross referencer, a monitor, and an assertion checker).

Manuscript received March 10, 1986; revised August 24, 1987 and August 2, 1988.

K. Shannon is with the Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599.

R. Snodgrass was with the Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599. He is now with the Department of Computer Science, University of Arizona, Tucson, AZ 85721.

IEEE Log Number 8930514.

The problem addressed by this paper is how to map the IDL structure specifications into data structure declarations of a particular target language, the C programming language [12]. The primary goals were as follows.

- The mapping should provide *language coverage*, in that it supports the full IDL language as detailed in its formal definition [19].
- The mapping should be *type safe*: violations of the IDL type model made by the programmer, when using the C types generated from the IDL specifications, should result in violations of the C type model and be flagged by the C compiler. At the same time, operations that are allowed in the IDL type model should be permitted by the C compiler. We effectively require a particular isomorphism between the two type models.

Several secondary goals for this mapping were also identified:

- The mapping should be *compile-time efficient*: the mapping should not impose substantial resource requirements on the C compiler.
- The mapping should be *runtime efficient*, both in space and time.
- The mapping should present a *good user interface* to the C programmer: the syntax for manipulating IDL data should be natural and not require knowledge of implementation details, the compile and runtime error messages should be informative, and the C representation should be alterable without necessitating widespread changes in the programmer's code.

Each criterion alone is easy to satisfy; addressing all five in a satisfactory fashion is surprisingly difficult.

There have been several attempts to extend the power of data declarations in existing languages (e.g., extensions to Lisp [20], to Modula-2 [27], and to C [33]). IDL differs from these efforts in that it is language-independent, requiring the target language interface designer to effect an adequate mapping of the IDL constructs onto the target language. There has been relatively little written on representations in target languages of IDL structures. General representation strategies have been suggested for representing IDL structures [8], [19]. The use of Diana as an intermediate representation [23], [36], variants of Diana [4], [9], and tools for using Diana [21], [22] have been investigated. Diana structures have been mapped to Ada in the Ada-In-Ada compiler [35] and to C in the Ada Breadboard Compiler (ABC) [37] and the Berkeley Ada Compiler [40]. The Diana Package [24] is the component of the ABC system that provides data types, data operations, and input/output functions for Diana. It is based on some of the implementation options given in the Diana Reference Manual (DRM) [8]. The major goal of the Diana Package was to quickly implement a prototype which realized the specifications in the DRM. In general, efficiency was not considered. The Berkeley Ada Compiler used the ABC to study implementation techniques for Ada, including an efficient runtime representation of Ada programs. The only change to the Diana Package was reducing the size of the Diana representation by using variable sized nodes rather than the fixed size nodes used in the ABC. In all three compilers, the representations were generated by hand rather than automatically as in our system.

PLUM is a package for managing abstract data types specified in a language resembling IDL [25], [26]. Its target language is C. It supports the dynamic definition of data structures, provides readers and writers, maintains a history record of modifications to instances, allows operations to be undone and redone using the history, and provides managers for significant events and for monitoring instances.

Finally, Graphite is a tool that takes a structure specification similar to IDL and produces two kinds of interface packages in Ada, one that permits specification modifications without requiring modification or even recompilation of user code, and one that is less flexible but is more efficient at run-time [5].

Our approach is similar to these other efforts in that we map an IDL specification to a target module, specifically, C macros, declarations, and functions. We insist on supporting the full IDL language and on ensuring complete type safety. Within those constraints, we then tried to make the mapping compile-time efficient, runtime efficient, and user-friendly.

The remainder of this paper is organized as follows. The second section discusses properties of the IDL language, emphasizing its type model. Further details on IDL may be found elsewhere [16], [19], [29]. Section III describes the C data structures generated from IDL structure specifications. The fourth section discusses type checking within the C representation. The last sections give an evaluation of the system, list future work, and propose an extension that addresses the remaining problems.

Throughout, we compare our implementation with the modified Diana Package (incorporating the change from fixed size to variable sized nodes) of the Ada Breadboard Compiler, with the Ada representation in the Ada-In-Ada compiler, with the proposed Ada package specification in the DRM, with PLUM, and with Graphite. For the remainder of the paper, we use "ABC" to refer to the modified Diana Package of the Ada Breadboard Compiler rather than to the compiler itself. IDL types and keywords are in a **boldface sans serif** font; C types are in a sans serif font.

II. IDL Specifications

IDL encompasses four aspects in its definition: constructs for specifying static data structures, constructs for specifying dynamic computation, constructs for specifying assertions, and a way to represent structure instances in a machine and language independent fashion. Only the first two aspects are relevant to the topic of this paper. We discuss each below, then examine the type model imposed by the language.

A. Specifying Data Structures with IDL

Structures are specified in IDL as directed graphs of attributed nodes. These structures encompass many of the data structures found in procedural programming languages, and are especially useful in specifying data structures employed by compilers. IDL provides four *basic types*, four kinds of *structured types*, and an escape mechanism (private types). The IDL basic types are named by the IDL keywords **Boolean, Integer, Rational,** and **String**. The IDL structured types are nodes (named collections of zero or more named values called attributes that the user wishes to treat as a unit), classes (a collection of node types sharing common aspects), sets (an unordered collection of object of a type; sets may not contain duplicates), and sequences (an ordered collection of objects of a type; sequences may contain duplicates). Attributes actually hold the data values; nodes are a grouping device. The domain of values that an attribute can hold is specified by its *type*. An attribute type can be a basic type, a structured type, or a node or class type. The IDL system automatically supplies run-time support routines to manipulate sets and sequences in accordance with their expected behavior. An example of a node declaration is shown below.

```
function =>name:       String,
           parameters:Seq Of formal_param-
                      eter;
```

Attributes having a node or class type allow directed graphs to be specified. Nodes can be referenced by several other nodes, permitting arbitrary sharing.

A *class* is a collection of nodes sharing common aspects. The elements of the class are called its *members*. An example of a class declaration is shown in Fig. 1.

The members of a class that are listed in its declaration are said to be *direct members*. In the example above, the "assignment" node, the "function" node, and the "loop" class are the direct class members of the "statement" class. The members of the "loop" class are also considered to be members of the "statement" class. These members, the "forloop" and "whileloop" nodes, are *indirect members* of the class.

The definition of IDL does not permit cycles in the class hierarchy; no class may be a direct or indirect class member of itself. However, a class may be the type for an attribute contained in the class, thus permitting recursive

```
statement            ::=   assignment | function | loop;
statement            =>    spos:  sourceposition;
loop                 ::=   forloop | whileloop;
booleanExpression    ::=   assignment | variable;
```

Fig. 1. Class type declaration.

definitions. The distinction is one of type versus instance: the structure (type) cannot exhibit cycles, but an instance of that structure, composed of nodes linked through their attributes, may form an arbitrary graph.

Attributes associated with a class are propagated to all members of the class. Such attributes comprise the common aspects shared by nodes of the class. These attributes will appear in the node and class members as if they were associated directly with the node or subclass. An example of this is shown using the statement class in Fig. 1, where the attribute spos is propagated to the assignment, function, forloop, and whileloop nodes. This allows the user to make a clear statement of the similarities among the member nodes of a class and helps avoid unintentional differences.

The class concept in IDL is similar to that of the same name in Simula-67 [2], Smalltalk [7], or C++ [33]. All support the definition of a hierarchical collection of classes (IDL and Smalltalk allow multiple hierarchies) with attribute inheritance down the hierarchy. They differ in that IDL is completely declarative, 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. This task is at best tedious and repetitive, if performance is not an issue, and at worst quite complex, if high efficiency is required.

Private types allow a special representation not supported by IDL to be used for a particular type. These types are effectively an escape mechanism to allow one to use specific representations in the target language. This concept is distinct from private types in Ada which allow the user to define an abstract type [41]. Six statements, in any order, are required [31]:

Type SourcePosition;
For SourcePosition **Use Package** position;
For SourcePosition **Use External Integer;**
For SourcePosition **Use Size** 32 **Bits;**
For SourcePosition **Use Alignment** 32 **Bits;**
For SourcePosition **Use Name** sourcepos;

The first statement declares the private type; the second specifies the module name ("position"); the third statement specifies the external representation of the type (in this case, as an integer); and the fourth and fifth specify the size and alignment of the internal representation. An optional sixth statement specifies the name of internal type ("sourcepos"); the declaration of sourcepos is provided in the position module. The module must define a function which maps an instance of the type in the external representation into the internal representation, and another function which maps from the internal representation into the external representation.

All node and class declarations in an IDL specification are grouped into named collections called *structures*. An example of a structure specification is

Structure Example **Root** Statement **Is**
. . .
End

The *root* of a structure is a node or class from which all other nodes and classes must be *reachable*, in that it must be possible to trace a path from the root node or class to all other nodes and classes declared in the structure. A path is traced through attributes of nodes and classes and through members of classes. At run-time, the root of a structure serves much like the root of a tree (although instances of structures are generally graphs). 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 the other nodes or classes that are reachable from the root.

B. Specifying Processes with IDL

A *process* is the IDL model for a computation. An instance of a process reads and writes instances of IDL-specified data structures through a collection of *ports*. There are two kinds of ports: **Pre** ports for input and **Post** ports for output. Each process manipulates a data structure termed the *invariant* that is the union of all port data structures used in that process. A target language and target machine must also be specified. Fig. 2 displays an example process.

Processes in IDL are similar to processes in the Unix model in which each input and output port is a stream. The major difference is that in IDL the streams are typed. Several enhancements to the IDL process model, proposed elsewhere [28], are beyond the scope of this paper.

C. The IDL Type Model

IDL is a strongly-typed language. The IDL type model places the following restrictions on the operations permitted on instances of IDL structures while they are being manipulated within a target language program.

• Only defined attributes of a node or class may be accessed. The type of the value of the attribute will be that specified by the structure definition. The defined attributes of a node (or class) are those associated directly with the node (or class) and attributes propagated from all classes having the node (or class) as a direct or indirect member. This restriction is similar to visibility rules in a statically scoped language.

```
Process ExampleProcess Is
        Pre Input: Example;
        Post Output: Example;
        Target Language C;
        Target Machine Vax;
End
```

Fig. 2. A Process declaration.

• A scalar or structured (set or sequence) typed attribute or variable may only be assigned a value of the same scalar or structured type. The only exception is that a value of type **Integer** may be assigned to an attribute or variable of type **Rational.** Arbitrary sharing of scalar or structured values should be supported. This restriction is present in most languages.

• An attribute or variable of a node type may only be assigned a reference to a node of the same node type. This restriction is also present in Oberon [38].

• An attribute or variable of a class type may only be assigned a reference to a node or class type which is a direct or indirect member of the class.

• The rules for assignment also hold for parameter passing during routine calls and returns.

• Operations such as append, isempty, etc. are supported for attributes or variables having a set or sequence type, with full type checking on the set or sequence elements. All operations for sets and sequences must preserve their semantics (i.e., sets do not contain duplicates, sequences are ordered).

The restrictions placed on scalar and structure operations are relatively easy to satisfy; supporting classes is more difficult. Two examples will illustrate some of the problems. If each node type is mapped into a distinct C type, then what should the C type of an attribute having an IDL class type be? The type rules state that nodes of various IDL node types, and hence C types, can be assigned to this attribute, and nodes of other IDL node types cannot be assigned. How do we arrange for the C compiler to allow some assignments but not others? The second example concerns compile-time versus runtime type checking. Occasionally type checking cannot be done completely at compile-time. How do we arrange for runtime type checking to occur only when dictated by the IDL type model? The next two sections show how these and other issues may be addressed.

III. The Target Data Structures

We have recently completed an implementation of an IDL translator that supports C and Pascal as target languages [29] and a set of IDL development support tools running on Unix (e.g., an IDL graphical printer [30]). The translator fully implements IDL, has been in active use since Spring 1985, and has been used to generate portions of several compilers, including itself, as well as other tools. The use of the IDL translator with C is illustrated in Fig. 3. In this figure, files are shown as rectangles and invocations of programs are denoted with ovals.

In our approach, the IDL translator generates two files for each process: one containing macro and data declarations, and a second containing code for the readers and writers declared in the process. In this paper, we focus entirely on the representation of instances of IDL structures in the C language. While the design and implementation of the readers and writers is certainly important [16], the topic is beyond the scope of this paper. This section describes the C representations chosen to implement the various data types defined in IDL. After describing our representation, we briefly discuss the representations supported by ABC, the Ada-In-Ada compiler, the DRM, PLUM, and Graphite. Type checking is discussed in the next section.

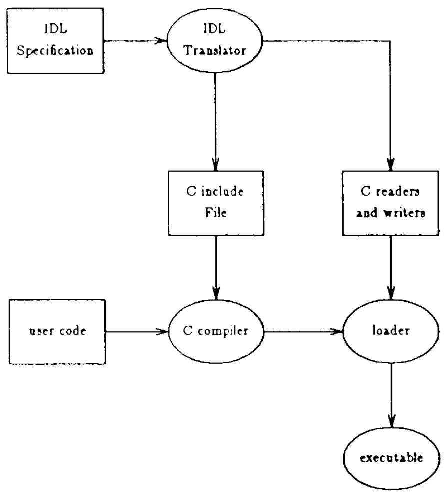


Fig. 3. Use of the IDL translator.

A. Scalar Data Types

IDL includes four scalar data types: **Boolean, Integer, Rational,** and **String.** A new C type named Boolean is provided to represent the IDL type **Boolean.** The C type Boolean is a typedef name for the C type char. The IDL data type **Integer** is mapped to the C type int. Representation specifications can replace the default mapping; the allowable alternative representations are signed char, unsigned char, signed short, unsigned short, long, and unsigned long. These user specifications provide control over space–time tradeoff decisions. The IDL data type **Rational** is mapped to the C type float. Alternatively, an object of type **Rational** can be represented as a C double.

Because C does not have a primitive string type, our interface provides a new C type named String to represent the IDL data type **String.** All Strings are maintained in a global String table. This representation guarantees that pointer equivalence is an adequate test for String equality. Macros are provided for converting from a String to a C pointer referencing a null-terminated array of characters, and vice versa. The latter conversion requires a search of the global string table. Efficient search is accomplished using a hashing function. The advantages are twofold. First, it saves both time and space as a result of reducing the number of dynamic memory allocation calls. Secondly, it greatly reduces the number of string comparison calls in a typical compiler since equality is based on pointer equivalance. Our representation of strings assumes that updating strings is less frequent than comparing strings. A simple analysis of the IDL translator and other compilers currently being implemented at UNC using IDL has shown that this assumption holds in this restricted domain. Invariant strings have also appeared in the Cedar [17] and GemStone [18] languages.

Conceptually, all scalar type objects of the same value are shared in our representation, in that value equivalence has the same semantics as object equivalence. Sharing of the scalar types **Integer, Boolean, Rational,** and **String** in our representation is more correctly defined as physical sharing rather than logical sharing, as with nodes, for performance reasons. Whenever two scalar type variables are equal, their values reside in the same memory location. This reduces the cost of comparison and eliminates the need for copying. Permitting arbitrary sharing of these objects, where modification of one object would modify the value of other variables sharing a reference to that value, would have incurred excessive space overhead (up to 100 percent, for integers, which would have to be allocated elsewhere). Additionally, arbitrary sharing of scalars is not required very often in compilers and other programming environment tools. While **String**s are implemented as references to arrays of characters, the representation ensures that references to equivalent strings will themselves be equivalent, as a performance optimization.

Representations for **Boolean, Integer,** and **Rational** in the ABC were int, int, and float respectively; no additional representations were provided. PLUM provides nine basic types, supporting Universal (an arbitrary 32-bit quantity), Univ_Ptr (a pointer to an arbitrary quantity), and Function_Ptr (a pointer to a function returning an integer) in addition to the C types listed above. Graphite supports all of the Ada predefined types, including Boolean, Integer, Float, and String, as well as private types. The treatment of IDL **String**s in the ABC is similar to ours. In the other systems, strings are pointers to an array of characters; no global string table is used.

B. Node Types

Two different node types are supported: attributed nodes (those with direct or propagated attributes) and unattributed nodes. Attributed nodes are represented by pointers to C struct types. The members of the C struct correspond to the attributes of the IDL node. The attributes are stored differently depending on their type. The scalar types **Boolean, Integer,** and **Rational** are stored directly in the struct. The scalar type **String** is represented as a pointer to an array of characters referenced by a global string table. Node types are represented by node pointers in the struct or as enumerated types. Classes, whose representation will be discussed shortly, are stored directly like the scalar types. Attributes with a set or sequence type are either stored inside the node or as a pointer depending on the Specified implementation. The struct includes a fixed header containing extra members generated for internal use by the runtime library routines. This information is stored in the first word of the struct and includes the type identifier (a unique even integer associated with each attributed node).

Unattributed nodes can be mapped to one of two different C representations. The default representation is identical to that for attributed noes. This representation allows several instances of an unattributed node to exist within an instance of an IDL structure and supports arbitrary sharing of these objects. A second representation for unattributed nodes is as odd integer constants, indicated with the following syntax:

For variable **Use Representation Enumerated;**

Conceptually this means that only one copy of each such type will be present within an instance of an IDL structure. The benefit, of course, is very fast "node" allocation.

When a node is created, either explicitly by the user or implicitly by the reader, all of its attributes are initialized to zero, corresponding to a false **Boolean** value, a zero valued **Integer** or **Rational,** an empty **String,** and an undefined node or class reference. The user can specify that a user-defined macro or function be called immediately after creation and initialization; this macro will normally perform additional initialization [31]. Local variables of type node or class must explicitly be assigned a node reference by the user.

Our approach for attribute storage is one of the options suggested in the DRM in that it utilizes the storage of attributes outside nodes (node references) and the storage of attributes inside nodes (scalar and enumerated types). In the ABC, nodes are implemented as C structs with attributes as members of the struct as in our implementation. The structure also consists of a fixed header of three words containing node kind, a unique node identifier used by the readers and writers, and a counter for marking. The main differences are that no attributes with a node type are stored inside the nodes (i.e., no enumerated types) and all nonscalar attributes are given an equivalent type, as described further in the next section. In the Ada-In-Ada compiler, nodes are represented as Ada variant records; the space overhead is unspecified. In PLUM, nodes are implemented similarly to the ABC, with an overhead of 2 words. PLUM restricts all attributes of the same name to have the same type. The representation of nodes in Graphite is not specified.

C. Classes

In our implementation, each IDL class is represented by a union of the node types that comprise the class. Information in a class is assigned and accessed through the class members except for common attributes of classes. An extra member exists in the union which is a pointer to a structure containing the common attributes of the class. Any of these attributes can be accessed through this field without knowing which node type is currently assigned to the class. This is possible because all of the attributes defined in a class are in the same position in every node belonging to that class.

Nonhierarchical classes complicate matters somewhat. With these classes, a graph coloring algorithm similar to one suggested for Diana [1] is used to order the attributes with a minimum waste of space. The difference is that our algorithm considers the size and alignment of the attribute when determining the order while the previous algorithm assumed attributes were of uniform size and alignment.

Two other representations for classes were also considered. The first was a flat representation where all class information was lost once the data structures were generated. This representation does not allow accessing attributes through classes, as class variables would not be supported. A second representation places pointers in nodes to their inherited attributes. This representation requires more space as well as an extra level of indirection to access attributes, without providing stronger type checking. Krogdahl has suggested yet another method: representing a reference of a class type with a pointer to within the struct representing the node [13]. Multiple inheritance in C++ is handled in the same manner [34].

In the ABC only one class is represented. A union containing a member for each node kind is defined. An attribute contained in a node structure is a pointer to this union. A generic structure is used to refer to any attribute of any node without regard to node kind or attribute name. Attributes are accessed using the appropriate offset. Their rationale for this representation was first that the sample package header in the DRM did this and second, that it made coding of the routines that used the ABC much simpler. In the next section, we discuss the disadvantages of this representation when considering issues of type safety.

The Ada-In-Ada compiler used a modified Diana consisting of 21 nonoverlapping classes, each containing from one to 31 node types. Attributes common to most or all nodes within a class were placed in the fixed part of the variant record.

PLUM and Graphite support a restricted version of classes, in that no attributes may be associated with classes. The representation is similar to ours: a union of the node types that comprise the class. Functions are provided for storing or retrieving values of attributes in a class.

D. Sets and Sequences

The IDL translator generates a representation and operation routines for each attribute type which is a set or sequence. Set and sequence components can be node or class types as well as scalar types. The C interface provides a default mapping for all component types. Alternative representations can also be specified. Reference [3] is a related approach, supporting stacks, sets, arrays, and the predefined types vertex, arc, and graph.

The representation for sets of **Boolean** is a C struct with two integer fields, true and false. The representation for sets of **Integer** is a bit vector. The size of the vector is by default the number of bits in an int (generally 32). Representation specifications for integer sets are used when a greater range is desired. The representation for sets of **Rational, String** and node or class types is by default a linked list of objects of the component type. All sequences are also represented in this manner by default. A linked list is made up of list cells containing two fields: the component object, and a pointer to the next cell in the list. Two types of array representations are also supported. For all representations, macros which invoke generic routines allow manipulation of sets and sequences without necessitating an understanding of the underlying implementation. For example, the following macros are generated for the type **Seq Of** A: inSEQA, initializeSEQA, appendfrontSEQA, appendrearSEQA, orderedinsertSEQA, retrievefirstSEQA, retrievelastSEQA, ithinSEQA, tailSEQA, removefirstSEQA, removeSEQA, removelastSEQA, copySEQA, foreachinSEQA, emptySEQA, equalSEQA, lengthSEQA, and sort SEQA. Only one group of macros is generated for each set or sequence with a particular component type. This means that the representation must be consistent for all sets and sequences with the same component type. An advantage of generating just one group of macros is that the representation can be changed without changing the programmer code. This makes it easy to experiment with different representations when trying to determine the most efficient. A disadvantage is that the programmer may want different representations for different attributes which have the same set or sequence type. However, experience indicates that these attributes usually have the same semantics and therefore, have similar operations performed on them. In such situations, there is no disadvantage to restricting them to the same representation.

The linked list representation is also used in the ABC and in PLUM. PLUM also supports sets implemented as bit vectors, but only for sets that may be represented in 32 bits. In the ABC, PLUM, and Graphite, sequences are untyped. No information on set or sequence representation was given for the Ada-In-Ada compiler.

E. Private Types

Private types are relatively easy to support, since the user must do most of the work, through user-supplied routines that create instances of private types, and convert such instances to and from their external IDL types. The system interfaces with the user through the input and output mapping functions. Other routines may also be required for initialization, finalization, marking, garbage collection, and move notification [31]. The graph coloring algorithm uses the specified size in its calculation of attribute positions.

PLUM supports a Universal, which is an arbitrary 32 bit quantity that can be used in place of a private type. In this system, the internal and external representations are both integers. Graphite provides support for private types of any size. The other systems do not support private types.

F. Processes

Currently, each process is mapped into a separate C program. Each **Pre** port is associated with a function of the same name that reads in an IDL instance from a file and returns the root of that instance. Each **Post** port is associated with a procedure that takes the root of an IDL instance and writes it out to a file. The readers and writers may be called multiple times. The invariant structure specifies all the possible attributes each node and class may have, and is the union of all the port structures.

For each process the IDL translator produces two files, an include file containing the data structure declarations and macro definitions, and the object code file containing the readers and writers of the process, to be linked in with the user's code comprising the algorithm for the process.

The ABC and PLUM do not support processes containing multiple ports associated with potentially different structures. The Ada-In-Ada compiler and Graphite do provide such support for processes, but do not support an invariant structure; the instances are required to be separate in main memory.

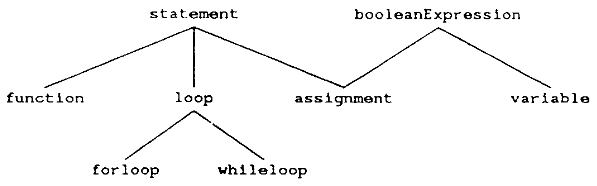
IV. Type Checking

The C language provides the structuring facilities of arrays, structs, pointers, and unions. The type model of IDL discussed in Section II is more flexible. In particular, attributes associated with classes can be accessed through the class, even though the node types may differ at runtime, and nodes of various node types can be assigned to variables of a class type. The challenge is to map IDL types into C types such that violations of the IDL type model by the programmer when using the C types result in violations of the C type model. At the same time, operations that are allowed in the IDL type model should be permitted by the C complier. Defining this kind of isomorphism between the type models is particularly difficult because the type models are so different.

This section describes the mapping from IDL types into C types. First, we give examples of using the macros provided by the interface. We next discuss the expansion of the macros to show how the implementation is used for type checking. Macros are used rather than direct manipulation in C of the internal data structures to ensure uniformity in the user interface and to hide some of the messy details. With macros, radical changes in the representation, such as storing attributes outside of the nodes in a separate data structure, do not require any changes to the user code. The complete IDL specification used in the examples in this paper, as well as the declarations and macros produced by the IDL translator from this specification, are available from the authors.

A. Selector and Conversion Macros

Static type checking is accomplished by the C compiler in conjunction with the data structures provided by the IDL translator. Dynamic type checking is accomplished by testing the instance of a class for validity. Macros provided by the interface facilitate both static and dynamic checking. For the macroexamples given below, we will use the example from Section II-A. Note that this is an incomplete example; many node types and attributes are omitted for the sake of brevity. This structure is illustrated graphically below.



We will assume that the following variables have been declared:

```
statement Astmt;
function Afunction;
forloop Aforloop;
loop Aloop;
assignment Anassignment;
booleanExpression AbooleanExp;
```

Note that variables of both node types and class types are present.

Attribute accessing is fairly straightforward and only requires static type checking. Functions are generated for valid attribute selections of nodes and classes. All of these functions are macros which require no procedure call overhead. The following are example expressions containing attribute selector functions which are legal to use.

```
nameOffunction(Afunction)
sposOfstatement (Astmt)
sposOfforloop (Aforloop)
```

The first is legal because the attribute name is associated directly with the node function. The second is legal because the attribute spos is associated directly with the class statement. The third is legal because the attribute spos is propagated to the node forloop from the class statement. These macros free the programmer from worrying about whether a value is a node or a class. Finally, the following is illegal

```
nameOfstatement (Astmt)
```

because the attribute name is not guaranteed to be in instances of the class statement. In this case, the macro would not be generated by the IDL translator, so that the C compiler can give an error message if the user accidently attempts to access this attribute through statement.

Assignments are more complex, because the types of both the left-hand side and the right-hand side must be taken into account. The C compiler does static type checking across assignment statements. For assignments of scalar and node types, this checking is sufficient. The following assignment statement, among many others, is allowed.

```
sposOfstatement (Astmt)
 = sposOffunction (Afunction);
```

For class types, however, more sophisticated checking is needed. When assigning a node to a class, the programmer must convert the node to the class (termed *widening*). This conversion requires static type checking; nodes may only be widened to classes containing that node type. A conversion macro is provided for each valid conversion of nodes to classes. An example is

```
Astmt =
assignmentTostatement (Anassignment);
```

When assigning a class to a node or to another class, the programmer must convert the initial class to the result node (termed *narrowing*) or result class (also considered narrowing if the result class is a subclass of the initial class). This conversion requires a runtime check to determine if the class contains a valid instance. The appropriate macro does both the conversion and the runtime check. Legal examples include

```
Afunction = statementTofunction (Astmt);
Aloop = statementToloop (Astmt);
```

In the first example, an error condition will occur if the value of the class variable Astmt is not a function node. In the second example, an error condition will occur if the value of the class variable Astmt is not a forloop or whileloop node. This is because these nodes are the only nodes that belong to both the statement and loop classes. Currently the runtime checking is implemented as an if statement; more efficient techniques are available, especially in portions of the class membership graph that are strictly trees [14].

Finally, some assignments require both narrowing and widening.

```
Astmt = assignmentTostatement (boolean
ExpressionToassignment (AbooleanExp))
```

In this example, the boolean expression variable is narrowed to an assignment node which is then widened to a statement.

For procedure calls, the conversion macros allow the expressions providing the values for the actuals to be converted to the appropriate type. The macros may also be used to convert the returned value to the appropriate type. The conversion in both cases is the responsibility of the caller.

The provided conversion macros guarantee that only valid node types can be assigned to a class. Stylistic conventions dictate that these conversion macros can only be used on the right hand side of an assignment statement. The C compiler will guarantee this by issuing an error message if the conversion macro is used on the left hand side of an assignment statement.

The programmer can determine which node is assigned to a class by using the provided macro typeof. This macro determines the node type given an instance of the class. Since the type information is contained inside the node structure, the type of the class returned by the typeof macro is guaranteed to correspond to the type of the actual node assigned to the class.

We now discuss the declarations produced by the IDL translator, especially the unobvious ordering required for type safety of the attribute accessing operations. We next discuss the expansion of the macros used above and how the C compiler is used to flag type violations.

B. Data Declarations

As mentioned in Section III-B, attributed nodes are represented by pointers to C struct types. The members of the C struct, prefixed with the letter "R", correspond to the attributes of the IDL node. The C structure for the function node is:

```
typedef struct Rfunction *function;
struct Rfunction { IDLnodeHeader IDLhidden;
   sourceposition spos;
   String name;
   SEQformal_parameter parameters;
};
```

The IDLhidden field contains extra members generated for internal use by the runtime library routines. This field is one word stored in the first word of every node structure that contains the type identifier and other information used internally by the runtime system.

The second field, spos, is a propagated attribute from the ancestor class, statement. This attribute is in the same position in every node belonging to the class statement. The remainder of the fields are attributes declared for the node function. All of the attributes are accessed through indirection. For example, for the following variable declaration:

```
function Afunction;
```

the attribute name would be accessed using:

```
Afunction->name
```

An IDL class is represented by a union of the node types that comprise the class. The members of the union are named by prefixing the node types with the letter "V". The declaration for the statement class is:

```
typedef union {
   int IDLinternal;
   Hstatement IDLclassCommon;
   assignment Vassignment;
   function Vfunction;
   loop Vloop;
   forloop Vforloop;
   whileloop Vwhileloop;
} statement;
```

The member IDLinternal is used to determine the type identifier of the node currently assigned to the class. If the value of IDLinternal is an odd integer, the class contains an enumerated type identified by this integer. If the value of IDLinternal is an even integer, the type of the node is determined by examining the type field in the IDLclassCommon member of the union;

```
#define typeof(c) ((!(c). IDLinternal) | |
((c).IDLinternal&1)?
  ((c).IDLinternal) : (c).IDLclassCommon->
  IDLhidden.TypeID)
```

It should be noted that this is not correct C code, in that the C language definition disallows examining the member of a union which is initialized to conatin a different member. In addition, the code assumes that a pointer requires as much space as an int, and that pointers are always even. Despite these problems, the code compiles and executes correctly on many machines, including the Vax! Since the target machine is specified in the process declaration (the declaration given in Fig. 2 specifies Vax as the target machine), the IDL translator can generate code and macros that are known to work for that target machine. The same comment holds for attribute placement: the attribute ordering algorithm must consider the alignment requirements of the target architecture and compiler when positioning the attributes.

The member, IDLclassCommon, is a pointer to a structure, prefixed with the letter "H", containing the common attributes of the class. This structure for the class statement is:

```
struct Hstatement {
  IDLnodeHeader IDLhidden;
  sourceposition spos;
};
```

The IDLhidden field is identical to the field contained in the node structures. The spos field is the class attribute propagated from statement. It is in the same position in every node belonging to that class. This allows the attribute spos to be access through the IDLclassCommon field of statement without knowing which node type is currently be accessed through.

C. *Expansion of Macros*

Accessing attributes is facilitated through the use of macros. An accessing macro is generated for each attribute of every node and class type. For example, the following attribute accessing macros would be generated for the node function:

```
#define sposOffunction(Afunction) Afunction
->spos
#define nameOffunction(Afunction) Afunction
->name
#define parametersOffunction(Afunction)
Afunction->parameters
```

The following macro would be generated for the class statement:

```
#define sposOfstatement(Astatement)
Astatement.IDLclassCommon->spos
```

However, as discussed previously, the macro

```
#define nameOfstatement(Astatement)
Astatement->name
```

is *not* generated because the attribute name is only in the class if it is assigned the node type function. Using an undefined macro is flagged by the C compiler in two ways. If the macro is used on the left-hand side of a C assignment, the C compiler will flag an illegal left-hand side and an illegal combination of pointer and integer. If the macro is used on the right-hand side, the C compiler will flag an illegal combination of pointer and integer. In addition, the loader will flag an undeclared procedure.

```
  25
  26 nameOfstatement(Astmt) = New-
  String("illegal");
  27      name = nameOfstatement(Astmt);
  28
"main.c", line 26: illegal lhs of assignment op-
erator
"main.c", line 26: warning: illegal combination
of pointer and integer, op =
"main.c", line 27: warning: illegal combination
of pointer and integer, op =
```

Two types of conversion macros are generated: widening and narrowing. Widening macros convert a node or subclass to an ancestor class. This conversion requires static type checking. Widening macros are generated for each direct or indirect ancestor of a node or class. For example, the node forloop would have the following widening macros generated for it:

```
loop IDLtemploop;
#define forloopToloop(Aforloop)
  (IDLtemploop.Vforloop = Aforloop, IDLtemp-
  loop)
  statement IDLtempstatement;
  #define forloopTostatement(Aforloop)
  (IDLtempstatement.Vforloop = Aforloop,
  IDLtempstatement)
```

Narrowing macros convert the initial class to the result node or or result subclass. This conversion requires a runtime check to determine if the class contains a valid instance. Narrowing macros are generated for each direct or indirect member of a class. For example, the statement class would have the following narrowing macros generated for it:

```
#define statementToassignment(Astatement)
  ((typeof(Astatement)= =Kassignment) ?
  Astatement.Vassignment :
  (ConversionError("statement","assign-
  ment"), Astatement.Vassignment))
#define statementTofunction(Astatement)
  ((typeof(Astatement)= =Kfunction) ? Astate-
  ment.Vfunction :
  (ConversionError("statement","function"),
  Astatement.Vfunction))
```

```
#define statementToloop(Astatement)
   (((typeof(Astatement)==Kforloop) ||
   (typeof(Astatement)==Kwhileloop) || 0)?
   Astatement.Vloop:
   (ConversionError ("statement","loop"),
   Astatement.Vloop))
#define statementToforloop(Astatement)
   ((typeof(Astatement)==Kforloop) ?
   Astatement.Vforloop :
   (ConversionError("statement","forloop"),
   Astatement.Vforloop))
#define statementTowhileloop (Astatement)
   ((typeof(Astatement)==Kwhileloop) ?
   Astatement.Vwhileloop :
   (ConversionError("statement","whileloop"),
   Astatement.Vwhileloop))
```

Use of the provided macros guarantee that only valid node types can be assigned to a class. The C compiler will flag invalid conversions in a C assignment statement:

```
   41
   42    Aloop = booleanExpressionToloop
   (AbooleanExp);
   43
"main.c", line 42: operands of = have incom-
patible types
```

The compiler will issue an illegal left-hand side message if the conversion macros is used on the left-hand side of a C assignment statement. This enforces a stylistic convention.

```
   38
   39    statementToforloop(Astmt) = Afor-
   loop;
   40
"main.c", line 39: illegal lhs of assignment op-
erator
```

The set and sequence macros are also arranged so that the C compiler will perform the adequate type checks. The following is one such macro.

```
formal_parameter IDLtempformal_parameter;
#define appendfrontSEQformal_parameter (for-
mal_parameterseq, formal__parametervalue)
   formal_parameterseq = (SEQformal_parame-
   ter)IDLListAddFront(
      (pGenList)formal_parameterseq,
      ((IDLtempformal_parameter = formal_pa-
      rametervalue), (*lin++)(&IDLtempformal_
      parameter)))
```

The first assignment statement will cause the type of the formal_parameterseq argument to be checked, and the second assignment (to the global variable IDLformal _parameter) will cause the type of the formal_parametervalue to be checked. The last portion is complex in part because C unions cannot be coerced.

V. Evaluation

In this section, we evaluate the mapping from IDL to C in terms of the goals set forth in the beginning of the paper: language coverage, type safety, runtime efficiency, compile-time efficiency, and the user interface. We also evaluate the ABC, the Ada-In-Ada compiler, the proposed Ada package specification in the DRM, and PLUM on these criteria.

A. Language Coverage

Our system supports all of the constructs of IDL, including the basic types, nodes, nonhierarchical classes, sets and sequences of any cardinality, and private types of any size. The one aspect not supported is arbitrary sharing of scalar values; we argued in Section III-B that such sharing is very expensive in space and time and is usually not needed. As noted in previous sections, ABC does not support nonhierarchical classes, private types, or arbitrary sharing of scalars. The Ada-In-Ada compiler does not support nonhierarchical classes or private types, and may not support sets or sequences or arbitrary sharing of scalars. DRM appears to support all IDL constructs. PLUM does not support arbitarily-sized private types, attributes associated with classes, or arbitrary sharing of scalars. Graphite does not support attributes associated with classes.

B. Type Safety

Our system supports the IDL type model discussed in Section II-C. First, only defined attributes of a node or class may be accessed. These attributes are accessed using selector macros. Second, node variables and attributes of nodes may only be assigned values of the same type. The exceptions are that a value of type **Integer** may be assigned to an attribute of type **Rational** (an implicit coercion allowed by the IDL type model) and that a value of type **Integer** may be assigned to an attribute of type **Boolean**, and vice versa. This latter coercion is included in the C language, and is carried over to the IDL interface because of its common usage. If strict type checking of booleans is desired, the following declaration can be substituted for the one given in Section III-A.

```
typedef struct { int value : 1 } Boolean;
extern Boolean IDLtrue = { 1 };
extern Boolean IDLfalse = { 0 };
```

A class variable or attribute having a class type may only be assigned a reference to a node or class type which is a direct or indirect member of the class. The C compiler will flag all compile-time violations of these assignments if the attribute accessing and type conversion macros are used. The rules for assignment also hold for parameter passing. In such cases, lint [11] will catch all violations if the appropriate macros are used. Runtime checks for assignments are provided by the appropriate macro. The narrowing macros do runtime checks only when necessary; however, they do not take context dependent information into account, such as in the following situation, where there is no need for runtime checks within the if statement

```
if (typeof(A) = = ...) ...
```

Also, the test is a sequence of binary comparisons, which could be made more efficient. Operations for attributes having a set or sequence type are implemented as macros which provide full type checking on the set or sequence elements. In addition, the translator only generates the operations which preserve the semantics of the set or sequence type. Finally, type checking of private types is fully supported.

This implementation has several advantages over the ABC in terms of type checking. In the ABC, a generic structure is used to refer to any attribute of any node without regard to node kind or attribute name. Attributes are accessed using the appropriate offset. In addition, all node or class attribute types are pointers to a union containing all the node types (a similar approach was employed manually in a Pascal application [10]). There are two problems with the implementation in the ABC. First, it requires the programmer to consider implementation details when accessing attributes. Second, it does not take advantage of type checking in the C compiler to flag invalid attribute selection and invalid assignments. Currently, the ABC only provides attribute selector functions for a few common attributes. These are implemented as macros. These macros are said to be error-prone and hard to maintain because they rely on the order in which the attributes are declared [24]. In our implementation, the selection of attributes through classes also relies on the attribute order but since the declaration file is generated automatically by the translator, there is no maintenance problem. Finally, the ABC only allows sequences of (untyped) nodes and does not support private types.

The Ada-In-Ada representation forces a tradeoff between class attributes, which appear in the fixed portion of the node, and node attributes, which appear in the variant portion. If the IDL specification is a strict, shallow class hierarchy, this representation works well (the modified Diana specification was a 2-level hierarchy). As previously mentioned, it cannot support the multiple hierarchies present in most IDL specifications nor does it support private types.

While the proposed package specification in the DRM is strongly typed in the Ada sense, it does not support the IDL type model at all. No type checking of either nodes nor basic types is performed; all values are of a private type "tree". Widening and narrowing are not supported, since classes in general are not supported. There is no type checking of sequences or sets or of private types.

Type checking in PLUM is quite simple, for two reasons: classes are not associated with attributes and all attributes of the same name must have the same type. An attributes in a node appearing as a value of a class variable is accessed through a function which looks up that attribute in the node. Hence C does the initial type checking, and the search for the attribute currently residing in a class variable is checked at runtime. PLUM does not do any type checking on narrowing nor on sequences. Sets are restricted to sets of integers, so type checking is trivial in that particular case. Private types are only supported through the Universal data type.

Graphite also does not allow attributes to be associated with classes. In all other cases, the target language Ada does the type checking, occasionally with some help from the user through explicit type specifications.

C. Runtime Efficiency

Our representation is resonably runtime efficient, both in space and time. Classes require no space overhead. The enumerated type representation for unattributed nodes requires space allocated only for an integer. The overhead for each node consists of one word. Attributed node references are simply pointers. Alternative representations for all the scalar types allow greater space efficiency. The representation is also reasonably time efficient in runtime time. Classes require no extra time at runtime. Runtime checking during attribute access or modification only occurs when required by the IDL type model; in particular, widening takes no time at runtime. Alternative representations for sets and sequences allows certain operations to be made more efficient.

In terms of space efficiency, the ABC differs from our approach in several aspects: enumerated types are not supported (hence, unattributed nodes still incur all the overhead of attributed nodes), the overhead per node is three times as great, sets and sequences must be represented as linked lists, and alternative representations for scalars are not available. The original implementation of the Diana Package was even worse, since it did not support variable length nodes. The ABC is quite efficient in time, since little is done at runtime. Widening and narrowing are not supported in the ABC. Enumerated nodes are also not present in the Ada-In-Ada compiler, which is otherwise quite efficient at runtime.

As no package implementation is given in the DRM, it is difficult to estimate runtime efficiency. Widening and narrowing are not supported.

PLUM is almost as space efficient as our approach. Since noninteger sets or sets of more than 32 integers are not allowed, most sets must be represented as linked list sequences. Attribute accessing and modification in classes is quite expensive, incurring the overhead of a function call and a hashed attribute search. As narrowing does not do any type checking, it is takes no time at runtime.

No package implementation is given for Graphite, so it is difficult to evaluate its run-time efficiency. However, since every access or modification involves a procedure call (perhaps an inline one) as well as a case statement on the attribute, these operations will probably be quite a bit slower than direct access, unless the Ada compiler performs very sophisticated constant folding.

D. Compile-Time Efficiency

The limitations in our implementation arise primarily in the areas of compile-time efficiency and the user interface. The current mapping results in a very large declaration file. Compiling source files with a large import environment is a common implementation problem termed *"big inhale"* [6]. For each node type there are four macros generated; allocation, initialization, deletion, and attribute deletion. For each **Seq Of** type there are eighteen sequence operation macros generated; for each **Set Of** type there are nine set operation macros generated. In addition, there are macros for selecting attributes of nodes and classes and for converting between nodes and classes. These macros contribute to the length and long processing time of the declaration file. For example, the Diana specification is 2585 lines long. The generated declarations file is 13,972 lines long, an expansion factor of over 5. Such an expansion is common.

The number of macros generated can be decreased with the **Restrict** clause of IDL notation (similar to the selective import used for the systems programming language MARY [6]). The generated operations are then restricted to only those used by the programmer. In our experience, use of this feature decreases the size of the declarations file considerably.

In the ABC and in PLUM, fewer macros are defined, because strong type checking is not supported (the ABC contains even fewer macros than PLUM). The preprocessor is still used heavily, and restrictions are not supported, resulting in lower compilation efficiency. The Graphite system is explicitly designed to reduce the need for recompilation when a specification changes. An ancillary benefit is that the generated Ada package is small, reducing compilation time when the compiler must be invoked. On the other hand, in Graphite, the Ada-In-Ada compiler, and the proposed package specification in the DRM, each attribute access invokes a (potentially inline) procedure, which requires a substantial amount of analysis at compile time or at runtime.

E. The User Interface

There are a few positive features of the user interface. First, automatically providing attribute selection functions, node and class conversion functions, and set and sequence operations frees the programmer from having to know the implementation details. Second, it is possible to change the representation of an attribute type without necessitating changes to the programmer's code. Finally, some of the conversion macros provide informative runtime error messages for invalid assignments of nodes and classes.

A limitation in the user interface is the procedural syntax required for selecting attributes and members of classes. The additional syntax often results in very long expressions when accessing through several levels of indirection. A second limitation is that the C compiler generates confusing error messages when a type violation is detected.

The other systems all provide a similar user interface, with analogous benefits and limitations.

F. Summary

In conclusion, the IDL translator and the C representation have been successful. The type checking required by the IDL type model is fully supported, in contrast to the other implementations, which fell far short of this goal. The runtime efficiency is also higher than the other implementations. In fairness, the other implementations did not have strong type checking nor runtime efficiency as primary goals. However, we feel that an acceptable implementation of IDL must fully support the type model and exhibit adequate performance.

The primary drawback to our implementation is the substantial compilation time incurred when using large structures such as Diana. A more powerful mechanism is needed to meet the compile-time efficiency and user interface goals we set for the mapping.

VI. Future Work

It is possible to meet the compile-time efficiency and user interface goals by using a special preprocessor rather than the standard C preprocessor as discussed above. We have designed a preprocessor that takes as input an IDL specification and a C program and generates a modified C program. The preprocessor is currently being implemented. The position of this preprocessor in the compile-load process is shown in Fig. 4, which should be compared to Fig. 3 in Section II-A. The user C code would look like valid C code, except that some variables would have IDL types rather than C types. Widening would be denoted by function call syntax, narrowing by a `switch` statement, and attribute accessing, whether through a class or a node, would be denoted by the familiar "->" construct. Set and sequence operations would be denoted by function call syntax, but with shorter names (e.g., `appendfront` rather than `appendfrontSEQformal_parameter`).

The preprocessor performs all the type checking required by the IDL type model. As a result, the representation seen by the C compiler is at a much lower level. A node is implemented as an array of words. An attribute of a node is accessed by using the appropriate index into the array. A class is just a pointer to a word. The preprocessor determines if a valid node instance is being assigned to the class. The preprocessor provides both static and runtime checking. The static checking involves determining whether the attribute being accessed can exist in the class. The runtime checking involves generating code to determine whether the attribute being accessed exists in the current instance of the node assigned to the class.

There are several advantages to using such a preprocessor. First, it does not require the use of a declaration

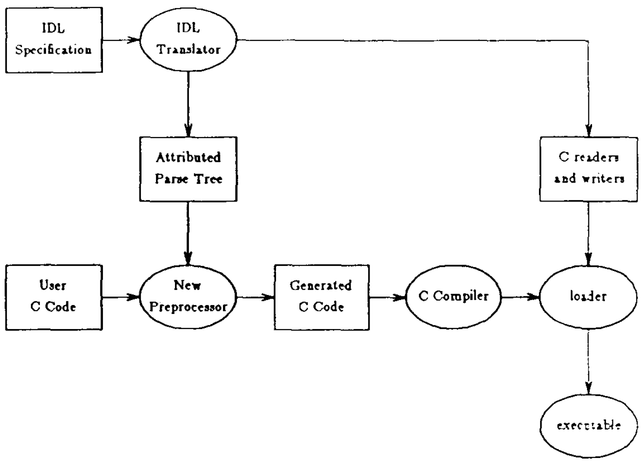


Fig. 4. Use of the new preprocessor.

file. This greatly reduces the processing time required to compile a source file using these definitions by eliminating the big inhale. Second, the procedural interface for selection can be discarded. No macros need to be generated for selections of attributes and class members. The preprocessor can determine if these operations are legal, inserting widenings automatically where necessary. Third, the preprocessor can provide better compile-time messages than are currently provided by the C compiler. Fourth, no changes to the C compiler are needed. Finally, runtime checking for assigning an instance of a class to another class is facilitated by providing code for runtime checks and forcing certain programming conventions such as explicit narrowing (typestate tracking is an especially appropriate technique in this context [32]).

There are also disadvantages. First, the preprocessor has to do both syntactic and semantic analysis of C programs. This is then redone by the C compiler. These tasks make the preprocessor complex and slow. A second disadvantage is the effort required for the implementation of the preprocessor, necessary for each target language supported.

There are several other important areas still to be investigated. One involves studying the interaction between the readers and writers of IDL instances and the representation of the invariant in C. This interaction presents new opportunities, such as memory partitioned by node type, and new problems, such as attributes in the invariant that are not present in the input structure. Another issue involves when the type checking is done: it can occur when the instance is read in, when an attribute is first accessed, or even when the instance is later written out. The performance ramifications of these alternatives are unknown.

Finally, we are interested in applying these techniques to other target languages. Initial study indicates that the techniques discussed in the first part of the paper do not apply to strongly typed languages such as Pascal and Modula-2, but that a partial solution is possible with Ada and Modula-2+ [27]. Languages, such as Oberon [38], that support type extension [39], appear to be compatible with these techniques. The preprocessor approach discussed in this section looks more promising with Pascal and Ada.

VII. Conclusion

Our goal was to develop a mapping from IDL structures to C that is type safe, compile-time efficient, runtime efficient both in space and time, and easy to use. Our solution is a translator which automatically produces C macros and data declarations. Through the use of the C data structuring facilities and macros, our system met the goals of language coverage, type safety and runtime efficiency without any of the limitations imposed by the other implementations examined. The goals of compile-time efficiency and good user interface were only partially met. A new preprocessor, currently being implemented, appears to achieve all five goals.

Acknowledgment

We would like to acknowledge T. Maroney, who implemented the parser and prototype code generator for the IDL translator, and who designed the initial mapping into C; J. Nestor and J. Newcomer, who provided valuable feedback and insight in many discussions; and the early users of our tools. D. Lamb and a refree provided valuable comments on a previous draft of this paper.

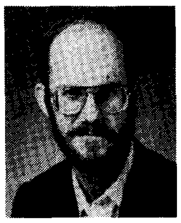
References

[1] A. Ambler, and R. Trawick, "Chaitin's graph coloring algorithm as a method for assigning positions to Diana attributes," *ACM SIGPlan Notices*, vol. 18, no. 2, pp. 37-38, Feb. 1983.
[2] G. M. Birtwistle, O. J. Dahl, B. Myhrhaus, and K. Nygaard, *Simula Begin*. Philadelphia, PA: Averbach, 1973.
[3] P. Breguet, F. Grize, and A. Strohmeir, "SARTEX: A programming language for graph processing," *ACM SIGPlan Notices*, vol. 20, no. 1, pp. 11-19, Jan. 1985.

[4] J. S. Briggs, "The design of air and its application to Ada separate compilation," in *Lecture Notes in Computer Science: Ada Software Tools Interfaces*, G. Goos and J. Hartmanis, Eds. New York: Springer-Verlag, 1983, pp. 60-75.
[5] L. A. Clark, J. C. Wileden, and A. L. Wolf, "Graphite: A meta-tool for Ada Environment Development, in *Proc. Int. Conf. Ada Applications and Environments*, Miami Beach, FL, Apr. 1986, pp. 81-90.
[6] R. Conradi, and D. H. Wanvik, "Mechanisms and tools for separate compilation," Univ. Trondheim, Norwegian Inst. Technol., Tech. Rep. 25/85, Oct. 1985.
[7] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*. Reading, MA: Addison-Wesley, 1983.
[8] G. Goos, W. A. Wulf, A. Evans, and K. J. Butler, *DIANA An Intermediate Language for Ada (Lecture Notes in Computer Science*, Vol. 161). New York: Springer-Verlag, 1983.
[9] M. P. Harrison, "Dianette—A pragmatic variant of DIANA," in *Lectures Notes in Computer Science: Ada Software Tools Interfaces*, G. Goos and J. Hartmanis, Eds. New York: Springer-Verlag, 1983, pp. 48-59.
[10] J. P. Jacky, and I. J. Kalet, "An object-oriented programming discipline for standard Pascal," *Commun. ACM*, vol. 30, no. 9, pp. 772-776, Sept. 1987.
[11] S. C. Johnson, "Lint, A C program checker," Bell Labs., Tech. Rep., 1982.
[12] B. W. Kernighan, and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.
[13] S. Krogdahl, "An efficient implementation of Simula classes with multiple prefixing," Inst. Inform., Univ. Oslo, Res. Rep. 83, Mar. 1985.
[14] D. Lamb, and R. Dawes, "Testing for class membership in multiparent hierarchies," Dep. Comput. Inform. Sci., Queen's Univ. Tech. Rep. 87-201, Nov. 1987.
[15] D. A. Lamb, "Sharing intermediate representations: The interface description language," Ph.D. dissertation, Dep. Comput. Sci. Carnegie-Mellon Univ., May 1983.
[16] —— "IDL: Sharing intermediate representations," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 297-318, July 1987.
[17] B. W. Lampson, "A description of the Cedar language: A Cedar language reference manual," Xerox Corp., Tech. Rep. CSL-83-15, Dec. 1983.
[18] D. Maier, J. Stein, A. Otis, and A. Purdy, "Development of an object-oriented DBMS," in *Proc. First Annu. Conf. Object-Oriented Programming Systems, Languages and Applications*, ACM, Portland, OR, Sept. 1986, pp. 472-482.
[19] J. R. Nestor, W. A. Wulf, and D. A. Lamb, "IDL—Interface description language—Formal description—Draft revision 2.0," Dep. Comput. Sci., Carnegie-Mellon Univ., Internal Document, June 1982.
[20] G. S. Novak, Jr., "Data abstraction in GLISP", in *Proc. SIGPLAN Conf.*, ACM, San Francisco, CA; June 1983, pp. 170-177.
[21] T. Payton, S. Keller, J. Perkins, and S. Mardinly, "The DIANA Interfacer," in *Lecture Notes in Computer Science: Ada Software Tools Interfaces*, G. Goos and J. Hartmanis, Eds. New York: Springer-Verlag, 1983, pp. 88-103.
[22] G. Persch, "The use of Diana in compilers, language transformers, formatters, and debuggers," in *Lecture Notes in Computer Science: Ada Software Tools Interfaces*, G. Goos and J. Hartmanis, Eds. New York: Springer-Verlag, 1983, pp. 76-87.
[23] G. Persch and M. Dausmann, "The intermediate language Diana: Design and implementation," in *Lecture Notes in Computer Science: Ada Software Tools Interfaces*, G. Goos and J. Hartmanis, Eds. New York: Springer-Verlag, 1983, pp. 23-34.
[24] M. E. Quinn, "The Ada breadboard compiler: The DIANA package," Bell Labs., Tech. Rep., 1982.
[25] S. P. Reiss, "An approach to incremental compilation," *ACM SIGPlan Notices*, vol. 19, no. 6, pp. 144-156, June 1984.
[26] ——, "PLUM—A data management package," Dep. Comput. Sci., Brown Univ., Tech. Rep., Apr. 1985.
[27] P. Rovner, "On extending Modula-2 build large, integrated systems," *IEEE Software*, vol. 3, no. 6, pp. 46-57, Nov. 1986.
[28] R. Snodgrass and K. P. Shannon, "Supporting flexible and efficient tool integration," in *Proc. Int. Workshop Advanced Programming Environments*, IFIP WG 2.4, Trondheim, Norway. New York: Springer-Verlag, June 1986, pp. 290-313.
[29] R. Snodgrass, *The Interface Description Language: Definition and Use*. Rockville, MD: Computer Science Press, 1989.
[30] R. Snodgrass, "IDL Toolkit Release Notes: Release 4.0," Dep. Comput. Sci., Univ. North Carolina, Chapel Hill, SoftLab Document, Mar. 1989.
[31] R. Snodgrass and K. P. Shannon, "Type extension in the Interface Description Language," in *Proc. Federal CASE Conf. Int. Data Management for Software Eng.*, Gaithersburg, MD, Nov. 1989.
[32] R. E. Strom, and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Trans. Software Eng.*, vol. SE-12, no.1, pp. 157-171, Jan. 1986.
[33] B. Stroustrup, *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1986.
[34] ——, "Multiple inheritance for C++," in *Proc. Spring 1987 European Unix Users Group Conf.*, Helsinki, May 1987.
[35] S. T. Taft, "DIANA as an internal Representation in an Ada-In-Ada Compiler," in *Proc. Ada TEC Conf. Ada*. ACM, Arlington, VA, Oct. 1982, pp. 261-265.
[36] J. Uhl, "A formal definition of DIANA," in *Lecture Notes in Computer Science: Ada Software Tools Interfaces*, G. Goos and J. Hartmanis, Springer-Verlag, 1983, pp. 35-47.
[37] C. S. Wetherell, "The Ada breadboard compiler: An overview," Bell Labs., Tech. Rep., 1982.
[38] N. Wirth, "From Modula to Oberon and the programming language Oberon," ETH, Tech. Rep. 82, Sept. 1987.
[39] ——, "Type extensions," *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 2, pp. 204-214, Apr. 1988.
[40] B. Zorn, "Experiences with Ada code generation," Univ. California, Berkeley, Tech. Rep. UCB/CSD 85/249, June 1985.
[41] *Reference Manual for the Ada Programming Language*, U.S. Dep. Defense, Washington, DC, ANSI/MIL-STD-1815A, 1983.

Karen Shannon is currently completing her Ph.D. dissertation at the University of North Carolina, Chapel Hill. She designed and implemented most of the IDL translator and many of the auxiliary tools in the IDL Toolkit. Her primary research interests include software engineering environments, programming languages, and compilers.

Ms. Shannon is a member of the Association for Computing Machinery and Sigma Xi.



Richard Snodgrass (S'79–M'81–SM'87) received the Ph.D. degree from Carnegie-Mellon University, Pittsburgh, PA.

He is an Associate Professor in the Department of Computer Science, University of Arizona, Tucson, and previously, at the University of North Carolina at Chapel Hill. His primary research interests are programming environments and temporal database management systems. He directed the implementation of the Interface Description Language (IDL) Toolkit and the implementation of the first temporal database management system. He is the author of a book on IDL.