IDL: Sharing Intermediate Representations

DAVID ALEX LAMB
Queen's University

IDL (Interface Description Language) is a practical and useful tool for controlling the exchange of structured data between different components of a large system. IDL is a notation for describing collections of programs and the data structures through which they communicate. Using IDL, a designer gives abstract descriptions of data structures, together with representation specifications that specialize the abstract structures for particular programs. A tool, the IDL translator, generates readers and writers that map between concrete internal representations and abstract exchange representations.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*modules and interfaces*; D.2.6 [**Software Engineering**]: Programming Environments; D.3.4 [**Programming Languages**]: Processors—*run-time environments*; E.2 [**Data**]: Data Storage Representations; H.2.3 [**Database Management**]: Languages—*data description languages* (*DDL*)

General Terms: Design, Performance

Additional Key Words and Phrases: Data representation, data structures, input/output, software engineering, system design

1. INTRODUCTION

A common theme of many approaches to improving programmer productivity has been increasing the use of tools. A single tool can increase productivity significantly, but complicated tasks might need a collection of tools. Thus integrating collections of tools has become important. This work addresses one aspect of this integration: controlling the exchange of structured data between different components of a large system.

The Interface Description Language (IDL) is a notation for describing collections of programs and the data structures through which they communicate. Using IDL, one can give abstract descriptions of data structures, together with representation specifications that specialize them for particular tools. I developed

This work was supported by the Defense Advanced Research Projects Agency, ARPA order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539, and by the Natural Sciences and Engineering Research Council of Canada under grant A0908.

Author's address: Queen's University, Department of Computing and Information Science, Kingston, Ontario K7L 3N6.

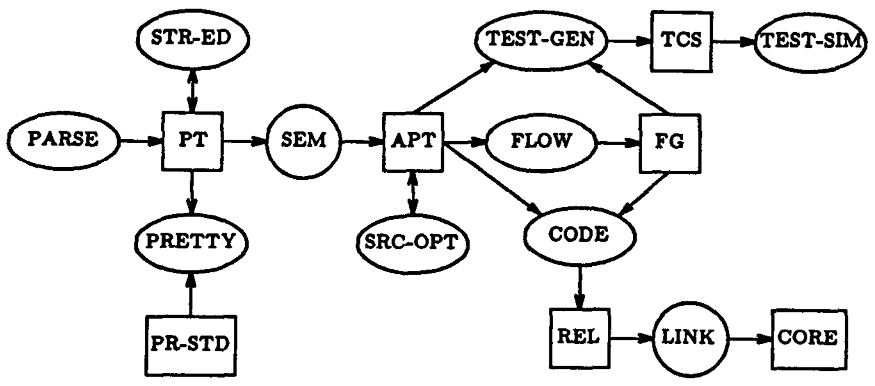

Fig. 1. Graphical system description.

IDL jointly with W. A. Wulf and J. R. Nestor [35] and investigated its usefulness in my Ph.D. dissertation [27].

The major claim of this paper is that IDL is a practical and useful tool for controlling the exchange of data among components of a large system. The remainder of this section outlines the goals we have for IDL. Section 1.3 describes how the rest of the paper demonstrates the major claim.

1.1 System Characteristics

IDL supports program collections like the one in Figure 1. Boxes represent data and ovals represent programs. A parser, PARSE, produces a parse tree, PT. A pretty-printer program, PRETTY, reads PT and produces a listing using conventions defined in a database called PR_STD. A screen-oriented language-based editor, STR_ED, operates on the parse tree and produces another parse tree. A semantic analyzer, SEM, generates an attributed parse tree, APT, from the simpler PT tree. Several tools operate on the attributed tree. FLOW creates a flow graph, FG. A source level optimizer, SRC_OPT, transforms an attributed tree into one representing an equivalent but more efficient program. A test-case generator, TEST_GEN, uses the attributed tree and the flow graph to produce a form, TCS, that can be used by the test-case simulator, TEST_SIM. Finally, a code generator, CODE, uses the flow graph and attributed parse tree to generate a relocatable file, REL. A linker, LINK, converts relocatable code into an executable core image, CORE.

This diagram illustrates three important ideas:

(1) The path PARSE to SEM to SRC_OPT to FLOW to CODE is much like the conventional breakdown of an optimizing compiler. Thus an aid for integrating tool collections would also aid in developing single programs, such as a multiphase optimizing compiler.

(2) Most of the data structures are complex, in that they require pointers of some kind. FG, for example, is a flow graph. If the tools do not simply communicate via shared memory, there must be some way to represent the data on external media.

(3) Components like TEST_SIM and LINK are obviously *reusable modules*. Any tool configuration can use them as long as it gives them appropriate input. This is also true of the other components of the diagram; SRC_OPT should work properly regardless of how its APT input was produced.

The last point leads to a slightly different interpretation of Figure 1. Instead of viewing the boxes like PT as data structures being read by particular programs such as SEM, one can view them as specifications of what such data structures must look like in order for the programs that read them to work correctly. Furthermore, a data structure like PT might be output from several programs and input to several other programs and thus is a specification of a common interface.

This example contains a variety of programs with different behavior. An editor might run on a dedicated workstation, whereas an optimizer might run on a mainframe with large amounts of memory. To be useful, each program must run efficiently and thus might need to have its data structures tuned to its processing requirements.

Thus these systems have four major characteristics:

(1) The programs are large.
(2) They communicate via complex data structures.
(3) They might be written in different programming languages and might run on different hardware.
(4) The internal data representations must be tailored to the program.

Later sections show how IDL supports these characteristics.

1.2 Tool Integration

One can integrate programs in a system at several levels. At the simplest level, each handles a single problem, ignoring other programs. At a higher level, the programs share some simple assumptions; UNIX[1] is an example of this. Many of the UNIX tools follow the *filter* paradigm, taking a single input and transforming it to produce a single output. A user forms higher level tools by concatenating filters using the UNIX command language and pipe mechanism. The tools share a simple model of their data: ASCII character streams. Conventions impose some higher level structures, such as text lines. This approach can be useful even in a non-UNIX environment [26].

Recently there have been attempts to design systems that achieve a higher degree of integration. In the Gandalf programming support environment [16], the tools share a representation based on the abstract syntax of the programming language being supported. In the Stoneman requirements for Ada[2] programming support environments, tools communicate via a shared database [6].

The UNIX environment integrated tools by imposing the simple character-stream model on all tool communication. The Gandalf and Stoneman environments use representations with richer, more complex structure. They integrate

[1] UNIX is a trademark of AT&T Bell Laboratories.

[2] Ada is a registered trademark of the U.S. Government (OUSDRE-AJPO).

tools by requiring designers to write them in a single language (C for Gandalf, Ada for Stoneman). This greatly simplifies the problem of exchanging data between the tools, since the same set of low-level internal representations are available to all tools. In addition, Gandalf uses a single data representation in all tools, which eliminates the need to transform the data structures when switching tools.

IDL achieves the next level of integration. Different tools use internal representations tailored to their needs but can exchange data with other tools.

1.3 Approach

The thesis of this paper is that IDL is a practical and useful tool for controlling the exchange of structured data between different components of a large system. The content of this paper is a first-level demonstration of this claim: It shows how to use IDL in a large system and provides performance results to support the claim that generic IDL-based tools are efficient enough for production use. Section 2 describes the IDL approach to designing large systems. Section 3 shows performance results for the prototype IDL tools (graph readers and writers) that I developed for my thesis system. Section 4 discusses related work showing how others have tackled the issues IDL addresses; it comes late in the paper to allow comparisons with IDL. Section 5, especially Sections 5.1 and 5.2, point to other demonstrations of IDL's usefulness that are beyond the scope of this paper.

2. IDL

IDL provides one solution to the problem of interconnecting large program components. The solution has three parts:

(1) A *notation* for describing systems and the data structures with which they communicate.
(2) A *program organization* imposed on program components of the systems for permitting them to communicate.
(3) A *translator* that analyzes the description of a system written in the IDL notation and generates fragments that plug into the program organization.

2.1 Program Organization

IDL imposes a particular organization, shown in Figure 2, on the program components of a system, which it calls *processes*. A process reads input data structures and produces output data structures. The diagram shows one of each, but some processes will have more. Non-IDL inputs or outputs (such as textual source or listing files) do not count; the parser of Figure 1 had no IDL inputs. For each IDL input there is a *reader* that maps the input data structure from an exchange representation to an internal representation. For each output there is a *writer* that maps the internal representation to an exchange representation. The code for the process accesses the internal data structure via an *interface*. The reader, writer, and interface can be generated from the IDL description of the process and its data structures.

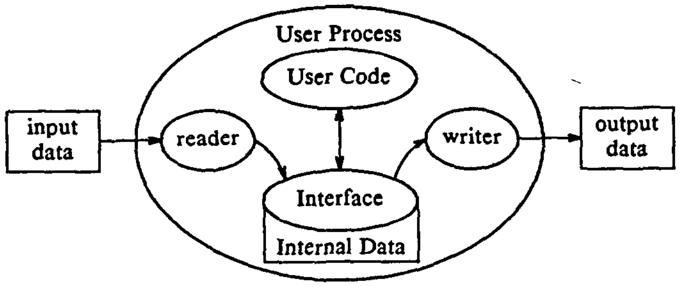

Fig. 2. Typical IDL process. Arrows indicate direction of data flow.

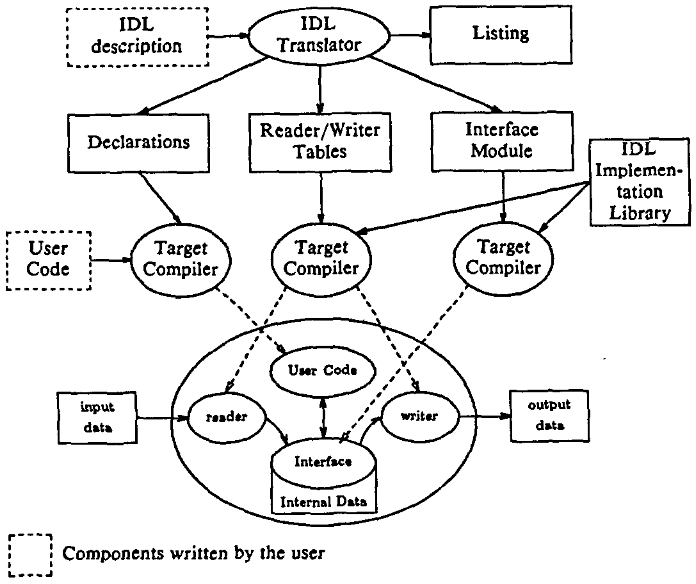

Fig. 3. Generating a user's process.

2.2 The Translator

The IDL translator is the primary tool of the IDL approach. It takes an IDL description and produces data declarations and module elements in a variety of target languages. It behaves as a compiler for the IDL notation.

The module elements plug into the program organization. They address several pieces of the data sharing problem.

—The reader/writers are tools that translate between the IDL external representations and particular internal representations used by individual processes.

—The interface modules define the details of the internal representations used inside the tools and provide access functions for manipulating this data.

Figure 3 shows the relations among these components. The system designer feeds an IDL description into the IDL translator, which produces a listing, declarations (e.g., an Ada package specification), an interface module (e.g., an Ada package body), and reader/writer tables. The listing contains a summary description of the interface module which aids a programmer in writing source code using the interface. This code is compiled along with the declarations produced by the translator to produce object code for the user's program. The interface module is compiled to give an implementation of the interface. The reader/writer tables are compiled along with structure-independent reader/writer skeleton modules from a library. The user code, reader, writer, and interface module are combined into a core image or load module by a linker (not shown). When it runs, the user process reads an input data structure, manipulates an internal data structure, and writes an output data structure.

The content of the reader/writer tables varies from implementation language to implementation language. The tables contain information on the layout of each node type, that is, what attributes the node contains, what their types are, and where to find them in the node. Some of this "tabular" information might be represented procedurally. For example, the tables might say that in order to store a particular attribute with a representation other than a simple bit string, the reader should call a particular procedure.

2.3 The Notation

With the IDL notation, a system designer describes the components of his system (via *process* declarations), the data structures by which the components communicate (via *structure* declarations), and some of the properties of the data structures (via *assertions*). This paper focuses on structure declarations. Warren et al. [45] give a tutorial introduction to using IDL.

Two principles guide the IDL notation:

(1) Separate abstract properties of data from representational details, according to the method of abstract data types.

(2) Organize structures into a hierarchy of *refinements*, with lower levels containing more implementation details than higher levels.

IDL models data as typed, attributed, directed graphs. This is a general model and does not unduly restrict the data one can describe with IDL. Nodes in a graph have types; a node's type determines what attributes it has. Attributes also have types; base types are integers, Booleans, character strings, and rationals (including all the typical fixed and floating point types). The four type constructors are sets, sequences, nodes, and classes. A class is a union of several node types.

Figure 4 shows an IDL declaration for a structure named *simple*. The class *tree* contains node types *leaf* and *inner*. *Leaf* nodes contain a single string-valued attribute. *Inner* nodes contain an operator field and a sequence of sons. Sons may be any tree nodes, that is, a mixture of leaf nodes and inner nodes.

Operator is a class consisting solely of nodes with no attributes; it thus serves the same purpose as an enumeration type. We borrowed this technique from the Ada Formal Definition [25]; it is useful, since at one stage of processing one might only distinguish among node types, while at some later stage one might

```
Structure simple Root exp Is
  tree ::= leaf | inner;
  inner => op: operator,
           sons: Seq Of tree;
  leaf => name: String;
  operator ::= plus | minus | times | divide;
    plus =>; minus =>; times =>; divide =>;
End
```

Fig. 4. Simple IDL structure.

```
Structure rep1 Refines simple Is
  For leaf.name Use fixedstring 20;
  For operator Use enumeration;
End
Structure rep2 Refines simple Is
  For leaf.name Use index;
End
```

Fig. 5. Refining structures.

need to add attributes. The earlier stages might use a space-efficient enumeration type, whereas later stages would need some full-fledged node representation.

IDL requires that any data structure have a root node from which all others may be reached by traversing structural attributes (such as *sons* in Figure 4). The readers and writers require a place to start their traversals. This is not a restrictive requirement; most data structures such as parse trees and flow graphs already have a well-defined root. If not, the designer can add a dictionary node that points to the roots of disjoint subgraphs. *Simple*'s root is a node of type *exp*.

The **For** clause permits representation specifications of particular attributes or types. At present, IDL does not prescribe a particular set of representation specifications; an IDL translator might support many well-known representational techniques. For example, sequences might be represented as singly linked lists, doubly linked lists, threaded lists, arrays, stacks, and so on.

Figure 5 shows new structures that refine the one of Figure 4 by adding **For** clauses. The new structures do not name root types; they are the same as that of the original structure. In rep1, fixed-length arrays of 20 characters represent names in leaf nodes; an enumeration type represents operators. In rep2, indices into a table of strings represent names, so that there is only one copy of any given string. The latter representation causes difficulties in prior systems because the indices are "disguised pointers." The reader/writer tables mentioned in Figure 3 contain information that lets the IDL system find all such pointers; the implementation library provides operations that let the reader/writer treat disguised pointers as though they were real ones. The details are the subject of a separate paper [28]. A process using structure rep1 may still communicate with one using rep2 but must translate its internal representation into an *exchange representation* (Section 2.4) to do so.

A more detailed representation of a structure might need to add attributes. For example, with

```
Structure rep3 Refines simple Is
  tree => depth: Integer;
End
```

all tree nodes have an integer-valued *depth* attribute. This attribute records the distance of the node from the root of the structure; the system designer could say so with the assertion sublanguage of IDL. This example also declares that all members of class exp have a depth attribute. This is preferable to the alternative

```
Structure rep3 Refines simple Is
  inner => depth: Integer;
  leaf => depth: Integer;
End
```

when there is an inherent reason for all nodes in a class to have a particular attribute. In the alternative, one does not know if the depth field of inner nodes has any relation to the depth field of leaf nodes, and one has no way to refer to the depth field of a tree node without first deciding whether it is an inner or a leaf node. Also, the preferred form requires fewer declarations.

As Figure 3 implies, the IDL translator has a large library of possible representation strategies for types available. For implementations that go beyond those stored in the library, a designer declares a *private type*, which in IDL terminology simply means a type whose detailed structure is unknown to the IDL system. The designer must write an implementation package for such a type; this package must provide operations that the reader/writer needs to do its task. Details of how the IDL system uses the implementation library, and how it interacts with private types, are the subject of a separate paper [28].

Figures 4 and 5 show simple tree structures. DIANA [13] is an example of a complex structure with shared information; for example, nodes representing uses of identifiers point back to nodes representing their defining occurrences. IDL also aids in changes of representation and communication between programs written in different languages. A Pascal program produced the data files used in the performance measurements of Section 3. I wrote the measured program in BLISS-36 [11], a low-level untyped system implementation language. These two languages have different ideas of data; in particular, the Pascal sequences were linked lists, whereas the BLISS sequences were fixed-length arrays.

2.4 Exchange Representations

If two programs wish to communicate, they must share the same abstract description of their data. If their representations differ, they must communicate via some exchange representation that bridges the two internal representations. To allow maximum portability of data structures, IDL defines a scheme for representing graphs in ASCII. The scheme is also human readable, so that a debugger might use it to display data structures. The representation scheme for a structure can be derived directly from its structure declaration.

The external representation of a data structure consists of a reference to the root node followed by a sequence of labeled nodes. The representation of a node consists of the name of the node type, followed by a list of attribute-value pairs. Any value may be labeled; all labels must be unique.

Figure 6 shows a diagram of a simple tree representing the expression (3.0 + .001)/5.0. Figure 7 is a nested-form external representation; Figure 8 is one of many possible flat representations. A reader must support both styles of representation, but a writer might implement one or might give a system

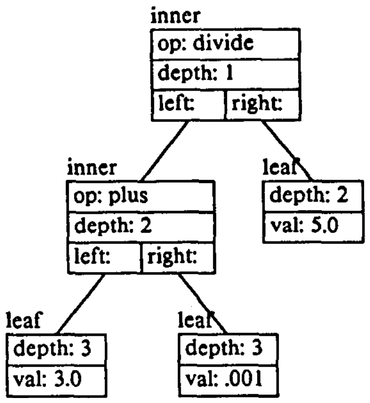

Fig. 6. Diagram of a simple expression.

Fig. 7. Nested external representation.

```
-(3.0 + .001)/5.0
inner [op divide; depth 1;
  left inner [op plus; depth 2;
    left leaf [val 0.3E1; depth 3];
    right leaf [val 1/1000; depth 3]];
  right leaf [val 5.0; depth 2]]
```

```
-(3.0 + .001)/5.0
L1: inner [op divide; depth 1; left L2^; right L3^]
L2: inner [op plus; depth 2; left L4^; right L5^]
L4: leaf [val 0.3E1; depth 3];
L5: leaf [val 1/1000; depth 3]];
L3: leaf [val 5.0; depth 2]
```

Fig. 8. Flat external representation.

designer a choice. The label-and-reference technique of this figure (Figure 8) also permits cross links within the data structure, which allows sharing of some attribute values. These figures also show that the exchange representation permits several forms of rational literals, which allow precise representations of any fixed-point or floating-point numbers.

3. RESULTS

The performance of the graph readers and writers of Section 2.4 has a direct bearing on the practicality of IDL. The main issues are

—Is the general-purpose reader/writer fast enough to be usable in a production environment?

—What do the trade-offs between performance and independence from specific representations look like?

I investigated several different versions of the reader/writer, varying from the general ASCII reader/writer to a hand-coded machine-specific version carefully tailored for a particular data structure. The general reader/writers are slower, whereas the faster ones lack portability. Most of the reader/writer packages interpret tables and code generated by the IDL translator. Therefore a designer can choose an appropriate point on the generality-versus-performance trade-off without giving up the benefits of automatic generation of reader/writers.

3.1 Reader/Writer Speed

The generic reader/writers impose overhead on doing graph input/output, since they can handle general graphs with arbitrary cross references. To determine the extent of this overhead, I compared four versions of the reader/writer.

(1) Version 1 is a generic ASCII reader/writer.

(2) Version 2 is a binary reader/writer that eliminates string processing overhead; it represents tokens by fixed-length binary encodings instead of variable-length ASCII strings.

(3) Version 3 is a hand-tuned reader/writer designed specifically for the tree structure used in the experiment. Each node is represented in a Polish form: a small integer for its type, followed by the representation of its descendants in a fixed order.

(4) Version 4 is the Linear Graph Package reader/writer from the Production Quality Compiler-Compiler (PQCC) project. The external form it produces is similar to a flat IDL external representation with minor syntactic differences.

Versions 1 and 2 are fully general graph reader/writers. Version 4 can also handle graph structures but it supports fewer attribute types; the experiment used attribute types in the common subset. All three permit reordering of fields during input and allow a program that expects more fields than are present in the input to read a graph, if the reader can supply simple defaults (such as a constant zero value) for missing fields. Version 3 maximizes speed and has none of this flexibility.

All four processed a simple tree structure, shown in Figure 9, where each node type has a fixed number of descendants. This is the worst case for a graph reader/writer, since the generality needed for cross links is unnecessary. I wrote all four in BLISS-36 and compiled them in nondebugging mode; this permits the BLISS-36 compiler to optimize the code. All used the same storage allocator, which was not tuned to the node sizes found in this application. I did not tune either the ASCII or binary reader/writers. The tests ran on a DECsystem20 2060 processor under TOPS-20. A 2060 executes about 2 MIPS. The BLISS-10/ BLISS-36 timing package collected timing data [37].

To show the exchange of data between programs written in different languages with different notions of data types, I wrote a Pascal program that generates test cases. The 49 test cases ranged in size from 400 nodes to 10,000 nodes. The version 1 test program read the ASCII file written by the Pascal program and wrote a copy. The tests for versions 2, 3, and 4 proceeded in two steps. First a special data translation program read the ASCII files produced by the Pascal program and wrote a temporary output file containing the data in the form expected by the test program. Second, the test program read the temporary file and wrote a copy. Thus I did all four timings under the same circumstances.

Figure 10 shows results of the comparison. Raw data from the timing measurements give execution time as a function of number of nodes. I fitted a straight

```
Structure RWtest Root exp Is
  exp ::= leaf | arb | binary;
  binary => left:exp, right:exp;
  binary ::= plus | minus | times | divide;
    plus =>; minus =>; times =>; divide =>;
  arb ::= semicolon;
    semicolon =>;
  arb => sons:Seq Of exp;
  leaf => value:Integer;
End
```

Fig. 9. IDL structure for reader/writer test.

Test	Input time		Output time		File size
	msec	factor	msec	factor	bytes per node
ASCII	2.378 ± .003	1.17	1.434 ± .002	1.08	16.83 ± .05
Binary	.981 ± .002	.48	.360 ± .002	.27	13.92 ± .09
Hand	.111 ± .0002	.05	.051 ± .0003	.04	2.99 ± .02
LG	2.028 ± .007	1.00	1.322 ± .010	1.00	29.06 ± .14

Fig. 10. Reader/writer comparison. Times are shown as milliseconds per node and as a multiple of the time for the LG reader/writer.

line to the data by standard linear regression techniques. For each reader and writer, the figure shows execution time as the number of milliseconds to read and write a node and as a fraction of the time taken by the LG reader/writer. The plus-or-minus numbers (±) report 95 percent confidence intervals from a standard T test. Since I could only measure CPU time, I report file sizes to give some basis for estimating I/O time.

The ASCII reader is about 17 percent slower than the LG reader, whereas the writer is about 8 percent slower, which seems acceptable. The binary reader is about 2.5 times faster than the ASCII reader, and the binary writer is four times faster. This shows that small losses in transportability give respectable speedups. However, the hand-coded version of the reader is almost 9 times faster than the binary reader, and the hand-coded writer is more than 7 times faster than the binary writer.

If these relative timings are translated to an absolute scale, a processing time of 1 millisecond per node (roughly, the time taken by the binary reader) implies that input of 10,000 nodes would require about 10 CPU seconds. The PQCC project found that there were roughly 4 abstract syntax tree nodes per source line, so 10,000 nodes translates to 2500 lines, giving a rate of 250 source lines per second, or 15,000 lines per minute (W. A. Wulf, personal communication, 1982). To put this in perspective, 1000 lines per minute is a rough benchmark figure for how fast a compiler typically runs. If the original compiler passed an in-core intermediate representation between the front end and the back end, using the IDL binary reader/writer to communicate between the 2 phases would only slow the compiler down to 937 lines per minute. The ASCII reader/writer would slow it to 863 lines per minute. These numbers are rough but show that even the prototype reader/writer is nearly fast enough for production environments.

Scheme	Node size	Input time	Output time	Transfer time
	(bytes)	(milliseconds per node)		
ASCII	16.83	.404	.387	.0188
Binary	13.92	.334	.320	.0155
Hand	2.99	.072	.069	.0033
LG	29.06	.697	.668	.0324
per byte		.024	.023	.0011

Fig. 11. Low-level I/O timings. All times are measured in milliseconds.

3.2 Cost of Low-Level I/O

The timings of Section 3.1 omitted the cost of the lowest level character input/output routines and transfer time to the disk. These costs vary significantly from system to system. Figure 11 shows timing results for the low-level routines used in the experiment, along with estimated disk transfer rates. The LG node size is roughly 12 characters larger than that for the IDL ASCII representation, because the flat style it uses requires 2 6-character labels for every node: the label definition in the representation of the node, and the label reference in the representation of the parent of the node.

Disk transfer rates are difficult to estimate because of the effects of operating system policies and the properties of the output devices. The TOPS-20 system used in the experiment has RP06 disk drives, which have a peak transfer rate of 806 kilobytes per second, average rotational latency of 6 milliseconds, and an average seek time of 30 milliseconds. TOPS-20 attempts to keep a file on a single cylinder and to minimize seek time; the basic unit of information is a page of 512 36-bit words. If all the pages of an output file fit into a program's working set, they will all be written in a single operation. Thus the numbers in Figure 11 assume only one seek is needed. I estimated a per-node cost by calculating the transfer time for as many pages as needed to hold 10,000 nodes and divided this number by 10,000.

A comparison of the Transfer Time column of Figure 11 with Figure 10 shows that CPU time for formatting or parsing an external representation completely dominates I/O time.

3.3 Detailed Speed Comparison

The measurements reported in Section 3.1 showed large differences in speed between the generic binary reader/writer and the hand-coded version. Possible reasons for this difference include[3]

(1) The generic reader/writer represents nodes as a sequence of attribute/value pairs. The tailored version omits the overhead of attribute tags by using a positional encoding of attributes (Attribute tags).

(2) The generic reader/writer interprets a set of tables for deciding how to read and write the graph given to it, whereas the tailored reader/writer has these decisions represented by inline code (Table interpretation).

[3] Parenthesized remarks refer to lines in Figure 12.

Test	Time		File size
	msec per node		Bytes per
	Input	Output	node
Binary			
(a) original	.981 ± .002	.360 ± .002	13.92 ± .09
(b) no attribute tags	.773 ± .003	.298 ± .002	8.91 ± .12
(c) check for sharing		.055 ± .003	
Fast			
(d) original	.111 ± .0002	.051 ± .0003	2.99 ± .02
(e) interprets tables	.131 ± .0002	.076 ± .0001	2.99 ± .02
(f) tagged lexemes	.268 ± .002	.157 ± .002	8.91 ± .12
(g) use of REP nodes	.266 ± .001		

Cost differences		Input		Output		File size
		Time	%	Time	%	(bytes)
Attribute tags	a–b	.208	24	.062	20	5.01
Interpreting table	e–d	.020	2	.025	8	
Tagged lexemes	f–d	.157	18	.102	33	5.92
Detect sharing	c			.055	18	
REP nodes	g–d	.155	18			
Calculated difference		.540	62	.244	79	10.93
Measured difference		.870	100	.309	100	10.93
Residual		.330	38	.065	21	.00

Fig. 12. Detailed binary I/O comparison.

(3) The generic reader/writer represents values as tagged lexemes, whereas the tailored version omits tags (Tagged lexemes).

(4) The generic writer makes an extra pass to detect sharing (Detect sharing).

(5) The generic reader/writer has overhead in its handling to allow for private types and labels; the tailored version omits this overhead (REP nodes).

To determine the effects of these various differences, I implemented and timed several additional versions of the binary reader/writer:

—A version of the binary reader/writer that omits attribute tags and reads and writes attributes in a fixed order.

—A version of the binary writer that just checks for sharing without doing any output.

—A version of the "fast" reader/writer that interprets tables of the same sort as used in the binary reader/writer. This measures the effect of table-interpretation overhead.

—A version of the "fast" reader/writer that uses tagged lexemes.

—A version of the "fast" reader that creates and deletes representation nodes similar to those used to communicate between the binary reader/writer and private type modules.

Figure 12 shows the results. The top half reports the timings; the bottom shows what portion of the difference between the fastest and slowest reader/writers is attributable to the hypotheses outlined above. The Residual line represents the portion of the difference between the two readers not yet accounted for. The experiments account for about four fifths (79 percent) of the difference in speeds of the writers and about two thirds (62 percent) of the difference in speeds of the readers. An interesting thing about these numbers is that interpreting tables contributes little to the difference in speeds (2 percent for the reader, 8 percent for the writer). This shows that the design decision to build reader/writer skeletons that interpret tables was a good one.

3.4 Label Handling

The experiment described in Section 3.1 used pure trees as test data and needed no labels. To determine the cost of label handling, I measured three more versions of the binary writer. The first version writes the same nested representation as the generic binary writer but labels every node. The remaining two versions write a *flat* form of binary representation. Every node-valued attribute is represented as a label reference; every node is labeled. The first flat version visits tree nodes in postorder, thus causing all references to be backward references. The second flat version visits nodes in preorder, forcing all references to be forward references. The difference between the time for the first version and the original nested binary reader/writer gives the time to process a label definition; the difference between the first and second gives the time to process a backward label reference; the difference between the first and third gives the time to process a forward label reference.

Figure 13 shows timing results for these three versions, along with a repetition of the timings for the generic binary reader. These tests could not use the larger test cases from the experiment in Section 3.1 because of storage requirements for the label table; the number of nodes ranged from 400 to 6000. We can calculate low-level input/output costs by multiplying per character costs of Figure 1 by 4 for the label length. Thus the time taken by the low-level input routines for a label is .096 milliseconds per node, the time taken by the low-level output routines is .092 milliseconds, and the transfer time to the disk is .0044 milliseconds.

The time to handle label definitions, backward references, and forward references are comparable to the time to handle a node: 49, 30, and 88 percent, respectively. However, if cross links compared with nodes are infrequent, they will not appreciably slow down processing. Further, the timing figures for the flat forms may overestimate the label reference processing time, since the flat representations have a shape different from the original binary and labeled binary forms.

3.5 Future Measurements

The results of previous sections show that the IDL reader/writer prototypes of my thesis system are nearly fast enough for production use on the type of structures for which I performed the experiments, namely small near-tree structures. Further work is needed to determine the effect of node size on reader/writer times; I expect such results would show that timings vary linearly.

A more difficult problem is to predict the cost for structures with many cross links, such as DIANA representations of Ada programs. I have not had a DIANA-based Ada implementation available for study, so I cannot determine how

Test	Time	File size
	msec per node	bytes per node
Original binary	.981 ± .002	13.92 ± .09
Labeled	1.464 ± .006	17.79 ± .11
Flat backward	1.764 ± .007	21.79 ± .11
Flat forward	2.326 ± .048	21.79 ± .11
Label definition	.483	
Backward label reference	.300	
Forward label reference	.862	

Fig. 13. Label processing times.

frequent each node type is and cannot report what fraction of attributes are cross links. Few such links are forward links—typically just the links from defining occurrences of identifiers to their definitions. Also, few types of nodes need to be labeled; cross links point to identifier definitions and type definitions. There is no need to label expression nodes and statement nodes (except those that contain the defining occurrence of a label or an explicit reference to a labeled statement). These thoughts suggest that the input/output of DIANA structures may not be too costly, but we need hard data to be sure.

4. RELATED WORK

Previous sections have mentioned IDL as a notation for describing intermediate representations of programs, as a way of automatically generating graph readers and writers, as a way of specifying programs, and as a means of describing the interconnections between components of a system. This section discusses work that preceded IDL in each of these areas.

4.1 Direct Influences on IDL

The immediate ancestor of IDL is the Linear Graph package (LG) of C-MU's Production Quality Compiler-Compiler (PQCC) project [29, 47]. PQCC investigated the construction of optimizing compilers. LG had a generic graph reader/writer, a notation for describing nodes, and a program (the Require file generator) for generating data declarations and tables for the reader/writer. LG supported only one language, BLISS [48].

During 1979–80, Intermetrics, Inc. developed its own variant of LG to support several languages, including Fortran, PL/1, C, and Pascal [21]. Also during this period, the PQCC project published its representation scheme for Ada programs, TCOL.Ada [5, 38, 42]; reactions showed that the input language for the Require file generator was a poor way to communicate such descriptions to other people. In addition, there was some interest in trying to be able to communicate between the Gandalf project [16] and PQCC. Gandalf was written in C [41] and ran on a VAX-11, a machine with 32-bit words, whereas PQCC was written in BLISS and ran on a DECsystem20[4], a machine with 36-bit words. We developed IDL during

[4] VAX and DECsystem20 are trademarks of Digital Equipment Corporation.

December 1980 and January 1981 to address these concerns; its first application was defining DIANA [12, 13], an intermediate representation for Ada programs.

4.2 Intermediate Representations

Since the first intended use of IDL was for defining intermediate representations, we were influenced by prior work in this area. The term "intermediate representation" comes from compiler building; it means a program representation that is intermediate between source language and object code. Simple one-pass compilers do not build intermediate representations. More complex compilers typically use some form of Polish notation; ones that do optimization have typically used quadruples or triples [1, 14; (W. A. Wulf, personal communication, 1982)]. Some portable systems use the assembly language of an abstract machine. Examples include OCODE for BCPL [40] and P-Code for Pascal [2, 34]. Janus [7] is a family of abstract machine languages designed to be less language dependent; it was used in compilers for a variety of languages [18]. All these are linear representations and do not require general graph input/output.

Some recent compilers have used program representations based on abstract syntax trees. PQCC developed a family of program representations called TCOL. The Karlsruhe Ada compiler's AIDA representation [9, 39] and the DIANA representation for Ada programs [13] are both based on the abstract syntax of the Ada Formal Definition [25]; many semantic attributes are cross links to other portions of the tree. DIANA has attracted considerable attention; the Army and Air Force Ada Programming Support Environments (APSE) both base their program representations on DIANA [23]. In a few software houses it has become common to use tree-based representations (B. Brosgol, personal communication, 1982). The designers of the Ada Integrated Environment MAPSE pay considerable attention to the problems of efficiently reading and writing subsets of the graphs representing the objects in the environment [22].

In the Gandalf program development environment, all the tools share a common intermediate representation based on the abstract syntax of the programming language being supported [17, 31]. The representations sometimes contain cross links; for example, all uses of a variable might be chained together. While tools are running they share representations in core. Files contain trees in Polish form; they represent cross links by giving the path from the root of the tree to the reference node (R. Ellison, personal communication, 1982).

4.3 Input/Output of Graphs

A major part of this paper shows that graph reader/writers generated by the IDL translator are useful tools. This section describes other work on schemes for transmission of graph-structured data.

Atkinson describes a language for describing compact lists in a machine-independent manner [3]. This notation, also called IDL (for Intermediate Data Language), defines an exchange representation between programs running on different machines of a network.

Herlihy and Liskov analyze the problem of transmitting values of abstract types in messages [19, 20] in CLU (a strongly-typed language supporting information hiding [30]). In their scheme, the equation

$$T\$\mathrm{Transmit}(A) = T\$\mathrm{Decode}(XT\$\mathrm{Transmit}(T\$\mathrm{Encode}(A)))$$

defines the *Transmit* operation of an abstract data type *T*, where *XT* is an existing transmissible type. Similarly, in IDL a module implementing a private type must supply input (decode) and output (encode) transformation routines, and the reader/writer supplies a collection of representation types (*XT*). Since IDL might be used with target languages that lack garbage collection, private modules must also supply a deletion routine in order for the IDL graph writer to be able to delete the structure built by the Encode operation.

Herlihy points out the problems of preserving sharing across transmission. When the Decode routine receives a circular structure, it must be able to name an object before its value is available. Herlihy solved this problem for CLU by adding the notation of an *uninitialized* object; decode functions must not use the values of such objects but may use their names. IDL requires that it be possible to allocate a node and separately assign its components; this is a response to the same problem.

Nelson [33] mentions the difficulties of transmitting objects containing pointers across a network. In his work, the parameters to a remote procedure call must be encoded in a form suitable for transmitting across a network, a process Nelson calls *marshalling*. He gives examples of encoding particular pointer-containing objects and suggests that similar techniques can be used for other objects on a case-by-case basis. In Nelson's work, the remote procedure call is transparent; there is no constraint on transmitting subsections of an object. Consequently, when an object is transmitted, there is no way for the low-level routines to know if other data structures share portions of it. The IDL reader/writer, on the other hand, knows of any sharing in the objects it transmits.

4.4 Specifications

Section 1.1 mentioned that we could view the IDL description of a system as a specification. In the terminology of Guttag, Horning, and Wing [15], a specification language consists of a syntactic domain, a semantic domain, and a rule defining the mapping between the two. By this definition, IDL is clearly a specification language—even more so if one considers the assertion sublanguage of IDL, which is beyond the scope of this paper. The IDL notation for data structures is the syntactic domain, the formal object model defines the semantic domain, and the denotational functions of the formal model give the mapping [36]. There is not yet a formal model of IDL processes.

Many existing specification languages are intended for specifying small programs or modules. In contrast, the Vienna Definition Language (VDL) and Vienna Development Method (VDM) metalanguage have been used to describe large software systems.

The Vienna Definition Language [46] has been used to give operational definitions of programming languages and to write abstract interpreters. An operational definition treats a construct as a high-level instruction and defines an interpreter that executes it. VDL has several kinds of elementary object, as well as tree nodes (called *composite* objects). The null object is a tree node with no components; this structure is similar to IDL, in which structure definitions often include a special null node type.

The Vienna Development Method metalanguage [4] differs from VDL in that it is oriented toward giving denotational descriptions of large software systems. A denotational semantics assigns meaning to a construct by giving an abstract mathematical entity that models its meaning; the Ada formal definition [25] and the IDL formal model [36] use this technique. In the VDM metalanguage, the basic entities are sets, tuples (corresponding to IDL sequences), functions, trees, and maps. A map is a set of ordered pairs; a symbol table is a particular type of map. IDL currently lacks maps; they must be represented as sets or sequences.

4.5 Module Interconnection Languages

IDL descriptions specify a collection of programs and the data structures by which they communicate. DeRemer and Kron [10] define a module interconnection language as providing a notation for describing the components of a system and the *resources* they provide to other modules and require from other modules. IDL shares characteristics of such languages.

Ada [8] and Mesa [32] include some facilities for describing module interconnections. A module can export procedures, variables, types, signals (or exceptions), constants, and so on. Module interconnection languages such as the one in Gandalf [16] or in Tichy's thesis system [44] separate resource requirements from the module that provides the resource.

All these systems interconnect fine-grain components, such as modules, consisting of a few pages of source code. In contrast, IDL interconnects large components and thus does not deal with fine-grain interactions such as procedure calls and exceptions. Section 5.3 considers scaling IDL.

5. CONCLUSION

This work has provided a first-level demonstration that IDL is a practical and useful tool for controlling the exchange of structured data between different components of a large system.

The general-purpose ASCII and binary reader/writers are nearly fast enough for production use when handling pure trees. Processing cross links (label definitions and references) in a more general graph is expensive; each label definition and reference costs roughly the same time as processing a full node. Thus a nested external representation is more efficient than a flat one, since the nested representation minimizes the number of labels.

A family of binary reader/writers trade off flexibility against performance. Reader/writers (a) and (b) of Figure 12 interpret tables generated by the IDL translator and handle arbitrary graphs. If the system designer declares that the graph is a pure tree, versions of readers (c) and (e) through (g) can also interpret tables generated by the IDL translator. Thus a system designer can pick an appropriate level of performance without giving up the benefits of an automatic generation of reader/writers.

5.1 Effect of IDL on Industry

The usefulness of IDL is also shown by the extent to which industrial practitioners have adopted it to serve their own needs.

The first major use of IDL was the description of DIANA [13], an intermediate representation for use in Ada compilers. Softech and Intermetrics, the contractors for the Army and Air Force Ada environments, both use IDL structure descriptions for the intermediate representations used in their projects (B. Brosgol, personal communication, 1982; R. T. Simpson, personal communication, 1982). Intermetrics has found IDL structure descriptions to be a useful level of specification for their intermediate representations. The same higher level description can capture several different choices for lower level representations, all with different efficiency considerations.

Manufacturers such as Burroughs and Sperry-Univac are developing IDL translators to aid in their software development efforts (T. F. Payton, personal communication, 1982; W. C. Roos, personal communication, 1982). The Burroughs effort is part of a compiler-compiler project. Tartan Laboratories uses IDL not only for intermediate representations but also for other complex data structures such as symbol tables. In addition, it has specialized languages for other aspects of tool building; these languages all build on the data structure description portion of IDL (W. A. Wulf, personal communication, 1982). Intermetrics has built several compiler development tools on top of IDL. For example, BONSAI is a tool that aids in generating tree transformation phases; it uses IDL to describe the data structures [24].

The Software Engineering Institute at Carnegie-Mellon University is exploring IDL as a tool for interfacing large sets of software tools. The project brings together IDL developers and users from industry, academia, and government and is identifying areas of potential technology transition. Nestor and his co-workers expect to publish a book containing a revision of IDL and guidelines for developing IDL-based environments (J. R. Nestor, personal communication, 1986). An SEI-sponsored workshop at Kiawah, South Carolina, May 19–21, 1986 brought together several practitioners with IDL experience. One result of the workshop is that there are plans for a special issue of *SIGPLAN Notices* on IDL, which should appear in late 1987.

Snodgrass and his colleagues at the University of North Carolina in Chapel Hill have an extensive collection of IDL-based tools, including an IDL-to-C translator. They are preparing a book describing their tools (R. Snodgrass, personal communication, 1986).

5.2 Further Work

This paper demonstrated IDL's usefulness by showing how reader/writers aided exchange of data between different programs at acceptable cost, and by showing that IDL is in use in industry. Section 3.5 suggested additional performance measurements to explore the details of how I/O efficiency varies with the complexity of the data structure. The remainder of this section describes further work, some now in progress, to explore IDL's utility in managing large systems.

Further experience with using IDL to communicate data among real tools is needed to see whether IDL meets the long-term goals outlined in Section 1.2. Snodgrass and Shannon [43] have done some work along this line, and we suspect some industrial sites using IDL could provide interesting data. An important part of such work is controlling the packaging of tools. Initially, one might write a new tool as a separate program. Later, one might want to combine it with a related tool in the same core image, which allows a change of representation without going through file operations. Such a change should ideally require only a small change in the input to an IDL translator, with no change to the user-written code for the tool. I do not know of anyone who has explored this problem in detail.

The packages generated by an IDL translator for structures as large as DIANA are big enough to strain many compilers and similar tools built to handle hand-written packages rather than machine-generated ones. Part of the solution to this problem may be to use the IDL derivation and refinement mechanisms to tailor smaller packages that give restricted views of the information described by larger packages. As yet, there is little experience with these mechanisms. This work may overlap with the problem of views in database systems.

5.3 Scaling

IDL supports systems like compilers or programming environments in which the components are large. This suggests considering how well IDL can handle much larger or much smaller systems.

The data description portions of IDL scale well in both directions. It is easy to imagine using IDL to describe a complex database at one end of the scale or a collection of messages at the other. Such applications would need different styles of input and output than we have shown here, since in neither case would one likely transmit an entire structure at one time. Systems in which the nature of the data changes dynamically, such as some Artificial Intelligence applications, cause difficulties; IDL has a static view of data types.

Scaling the processes (i.e., program components) that can be described with IDL is a little more difficult. At the small end of the scale, instead of just exchanging data, processes might interact via procedure calls, access to variables, signaling each other, and so on. At the large end, it becomes important to describe processes composed of subprocesses; as it stands, an IDL description of a large programming environment cannot distinguish the set of processes making up a conventional compiler from the total set of programs in the whole environment.

REFERENCES

1. AHO, A. V., AND ULLMAN, J. D. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1978.
2. AMMANN, U. The method of structured programming applied to the development of a compiler. In *Proceedings of the 1973 ACM International Computing Symposium.* ACM, New York, 1974.
3. ATKINSON, M. P. IDL: A machine-independent data language. *Softw. Pract. Exper. 7,* 6 (Nov. 1977), 671–684.
4. BJORNER, D., AND JONES, C. B. *The Vienna Development Method: The Meta-Language.* Springer-Verlag, New York, 1978.
5. BROSGOL, B. M., NEWCOMER, J. M., LAMB, D. A., LEVINE, D., VAN DEUSEN, M. S., AND WULF, W. A. TCOL.Ada: Revised report on an intermediate representation for the preliminary Ada language. CMU-CS-80-105, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa. (Feb. 1980).
6. BUXTON, J. N. Stoneman: Requirements for Ada programming support environments. Defense Advanced Research Projects Agency, Arlington, Va. (Feb. 1980).
7. COLEMAN, S. S., POOLE, P. C., AND WAITE, W. M. The mobile programming system, Janus. *Softw. Pract. Exper. 4* (1974), 5–23.

8. DARPA. *Reference Manual for the Ada Programming Language.* ACM, New York, July 1982. (Order number 825820).
9. DAUSMANN, M., DROSSOPOULOU, S., GOOS, G., PERSCH, G., AND WINTERSTEIN, G. *AIDA Introduction and User Manual. 38/80.* Institut fuer Informatik II, Univ. Karlsruhe, Karlsruhe, West Germany (1980).
10. DEREMER, F., AND KRON, H. H. "Programming-in-the-large" versus "programming in the small". *IEEE Trans. Softw. Eng. SE-2,* 2 (June 1976), 80–86.
11. DIGITAL EQUIPMENT CORPORATION. *BLISS Language Guide.* DEC, Maynard, Mass., 1977.
12. GOOS, G., AND WULF, W. A. *Diana Reference Manual.* CMU-CS-81-101, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pa. (Mar. 1981).
13. GOOS, G., WULF, W. A., EVANS, A., AND BUTLER, K. J. DIANA: An intermediate language for Ada. *Lecture Notes in Computer Science 161,* Springer-Verlag, New York, 1983.
14. GRIES, D. *Compiler Construction for Digital Computers.* John Wiley, New York, 1971.
15. GUTTAG, J., HORNING, J., AND WING, J. Some notes on putting specifications to productive use. *Sci. Comput. Program.* (Oct. 1982), 53–68.
16. HABERMANN, A. N. The Gandalf research project. *Comput. Sci. Res. Rev.,* Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa. (1978–79).
17. HABERMANN, A. N., ELLISON, R., MEDINA-MORA, R., FEILER, P., NOTKIN, D. S., KAISER, G. E., GARLAN, D. B., AND POPOVICH, S. *The second compendium of Gandalf documentation.* Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa. (May 1982).
18. HADDON, B. K., AND WAITE, W. M. Experience with the universal intermediate language Janus. *Softw. Pract. Exper. 8,* 5 (Sept. 1978), 601–616.
19. HERLIHY, M. P. Transmitting abstract values in messages. MIT/LCS/TR-234, Massachusetts Institute of Technology, Cambridge, Mass. (1980).
20. HERLIHY, M., AND LISKOV, B. A value transmission method for abstract data types. *ACM Trans. Program. Lang. Syst. 4,* 4 (Oct. 1982), 527–551.
21. INTERMETRICS, INC. Intermetrics LS system description. IR 536, Intermetrics, 733 Concord Ave., Cambridge, Mass. 02138 (Aug. 1980).
22. INTERMETRICS, INC. Draft computer program development specification for the Ada Integrated Environment: MAPSE generation and support. IR 680, Intermetrics, 733 Concord Ave., Cambridge, Mass. 02138 (Mar. 1981).
23. INTERMETRICS, INC. Draft computer program development specification for the Ada integrated environment: Program integration facilities. IR 681, Intermetrics, 733 Concord Ave., Cambridge, Mass. 02138 (Mar. 1981).
24. INTERMETRICS, INC. Compiler retargeting tools user's guide. IR-MA-623, Intermetrics, 733 Concord Ave., Cambridge, Mass. 02138 (May 1986).
25. KAHN, G., DONZEAU-GOUGE, V., AND LANG, B. Formal definition of the Ada programming language. Honeywell, Inc., Cii Honeywell Bull., INRIA (Nov. 1980).
26. KERNIGHAN, B., AND PLAUGER, P. J. *Software Tools in Pascal.* Addison-Wesley, Reading, Mass., 1981.
27. LAMB, D. A. Sharing intermediate representations: The interface description language. Ph.D. dissertation, CMU-CS-83-129, Carnegie-Mellon University, Pittsburgh, Pa. (May 1983).
28. LAMB, D. A. Generating interface packages for IDL descriptions. Department of Computing and Information Science, Queen's University, Kingston, Ontario (June 1987).
29. LEVERETT, B. W., CATTELL, R. G. G., HOBBS, S. O., NEWCOMER, J. M., REINER, A. H., SCHATZ, B. R., AND WULF, W. A. An overview of the production quality compiler-compiler project. *IEEE Computer 13,* 8 (Aug. 1980), 38–49.
30. LISKOV, B., ATKINSON, R., BLOOM, T., MOSS, E., SCHAFFERT, J. C., AND SNYDER, A. *CLU Reference Manual.* Springer-Verlag, New York, 1981. Also appeared as MIT Laboratory for Computer Science, Tech. Rep. MIT/LCS/TR-225 (Oct. 1979).
31. MEDINA-MORA, R. Syntax-directed editing: Towards integrated programming environments. Ph.D. dissertation, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa. (Mar. 1982).
32. MITCHELL, J. G., MAYBURY, W., AND SWEET, R. Mesa language manual version 5.0. CLS-79-3, Xerox Palo Alto Research Center, Palo Alto, Calif. (Apr. 1979).
33. NELSON, B. J. Remote procedure calls. Ph.D. dissertation, CMU-CS-81-119, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa. (May 1981).

34. NELSON, P. A. A comparison of Pascal intermediate languages. *SIGPLAN Not. 14*, 8 (Aug. 1979), 208–213.
35. NESTOR, J. R., WULF, W. A., AND LAMB, D. A. IDL—Interface description language: Formal description. CMU-CS-81-139, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa. (Aug. 1981).
36. NESTOR, J. R., WULF, W. A., AND LAMB, D. A. IDL—Interface description language: Formal description. Computer Science Department, Carnegie-Mellon University (June 1982). Draft revision 2.0. Available from the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pa.
37. NEWCOMER, J. M. BLISS timer package user's manual. Computer Science Department, Carnegie-Mellon Univ., Pittsburgh, Pa. (1981).
38. NEWCOMER, J. M., LAMB, D. A., LEVERETT, B. W., LEVINE, D., REINER, A. H., TIGHE, M., AND WULF, W. A. TCOL.Ada: Revised report on an intermediate representation for the DoD standard programming language. CMU-CS-79-128, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pa. (June 1979).
39. PERSCH, G., WINTERSTEIN, G., DAUSMANN, M., DROSSOPOULOU, S., AND GOOS, G. AIDA Reference Manual. Nr. 39/80, Institut fuer Informatik II, Universitaet Karlsruhe, Karlsruhe, West Germany (Nov. 1980).
40. RICHARDS, M. The portability of the BCPL compiler. *Softw. Pract. Exper. 1* (1971), 135–146.
41. RITCHIE, D. M., JOHNSON, S. C., LESK, M. E., AND KERNIGHAN, B. W. The C programming language. *Bell Syst. Tech. J. 57*, 6 (July–Aug. 1979), 1991–2019.
42. SCHATZ, B. R., LEVERETT, B. W., NEWCOMER, J. M., REINER, A. H., AND WULF, W. A. TCOL.Ada: An intermediate representation for the DoD standard programming language. CMU-CS-79-110, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pa. (Mar. 1979).
43. SNODGRASS, R., AND SHANNON, K. Supporting flexible and efficient tool integration. SoftLab Document No. 25, Computer Science Department, University of North Carolina, Chapel Hill, N.C. (May 1986).
44. TICHY, W. Software development control based on system structure description. Ph.D. dissertation, CMU-CS-80-120, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa. (Jan. 1980).
45. WARREN, W. B., KICKENSON, J., AND SNODGRASS, R. A tutorial introduction to using IDL. SoftLab Document No. 1, Computer Science Department, University of North Carolina, Chapel Hill, N.C. (Dec. 1985).
46. WEGNER, P. The Vienna definition language. *Comput. Surv. 4*, 1 (Mar. 1972), 5–63.
47. WULF, W. A. PQCC: A machine-relative compiler technology. In *Fourth International COMPSAC Conference* (Oct. 1980), Appendix B, IEEE Press, New York, pp. 24–36.
48. WULF, W. A., RUSSELL, D. B., AND HABERMANN, A. N. BLISS: A language for systems programming. *Commun. ACM 14*, 12 (Dec. 1971), 780–790.

Received May 1985; revised August 1986; accepted September 1986