An IDL Evaluation

R. Clayton
Spencer Rugaber

Software Research Center
Georgia Tech

October 11, 1993

Abstract

The Interface Description Language (IDL) is designed to organize and support information exchange between software tools. We are implementing a system to perform source-to-source transformations on programs and have used IDL to create the intermediate representation upon which the transformations take place. This paper describes our experiences and observations about using IDL in the transformation system.

Introduction

A common way of organizing software systems is as a toolkit. Under the toolkit organization, the system appears as a set of programs (tools), each of which performs some simple task. Creating software to perform a more complicated task involves combining existing tools.

A requirement for toolkit organizations is a common intermediate representation for the data passed between tools. Without such a common representation, two tools would be unable to communicate since neither would be able to understand the data from the other. The Intermediate Description Language (IDL) is a method for generating intermediate representations for software tools.

This report describes IDL and and its use in a system supporting source-level program transformations. The first section describes IDL, the second section describes the work in which IDL was used, and the third section comments on how well the work went using IDL.

IDL

The Intermediate Description Language (IDL) is a language for describing and implementing data structures organized as graphs [Sno89]. A graph node represents a collection of related data. A directed graph edge from one node to another represents a reference from the *parent* node to the *child* node. Nodes and edges may be arranged to form arbitrary graphs, but nodes are typed and edges should point to only children nodes of the type expected by the parent node.

Nodes and Classes

An IDL *node* is similar to a Pascal record or a C struct. A node has a name and a set of zero or more *fields*, where each field has a name and a type. IDL provides the *atomic types* `Boolean`, `Integer`, `Rational`, and `String`. A node also defines an atomic type with its name being the type name. A node field with a node type represents a reference to a child node. The type constructors `Set` and `Seq`, when applied to an atomic type, result in a set and a double ended queue of values of the atomic type. `Set` and `Seq` can be applied only to atomic types.

A *class* is similar in structure to a node, having a name and a set of zero or more fields. A class's name also defines a type. In addition to named and typed fields, a class may contain unnamed references to other nodes or classes. Each reference establishes a *subtype* relation between the referencing class (the parent) and the referenced node or class (the child). The subtype relation is the usual one: a child, in addition to being of its own type, may also be considered to be of its parent's type. For the subtype relation to hold, children inherit their parent's named fields. Subtyping is transitive, meaning that all descendent of a class are subtypes of the class and that a child inherits all named fields in all its ancestors. A child may have more than one parent; that is, multiple inheritance is supported. The resulting type hierarchy is a graph with nodes at the sinks.

IDL assigns a data structure a type by selecting one of the component classes or nodes as the *root* of the data structure and giving the data structure the type of its root. An IDL *structure* defines the data structure's component classes and nodes and defines the root.

Processes and Assertions

IDL takes a sequential view of computation. A computation inputs a set of data structures and outputs some other set of data structures which are then passed on to the next stage of the computation. IDL represents a computation by a *process* and defines a process with two sets of named data structures, one set giving the computation's inputs and the other set giving the outputs. Each data structure defines a *port* for the computation; the port has the same name as the associated structure. A port is either input or output (but not both) depending on the direction the associated data structure moves in the computation.

IDL provides *assertions* to characterize a process's transformations between input and output data structures. Assertions are given as first-order predicate calculus formulas in process definitions. Assertions may also be included in structure definitions to specify data structure properties holding independent of computations. The non-procedural assertions do not generate any code for the process; they just characterize the computation's data structure transformations.

Mechanics and Use

IDL is designed to be language independent. Structures and processes are specified in IDL's own textual language and are translated into constructs appropriate to the host language. IDL currently generates C and Pascal code (Pascal will not be considered further in this note).

SCORPION is a suite of IDL tools [Sno91]. The major tool is the IDL compiler `idlc`, which takes structure and process definitions and generates the associated `.h` and `.o` files. The assertion checker `idlcheck` uses process and structure assertions to determine if a process has correctly manipulated its input and output structures. Other SCORPION tools include browsers, pretty printers, listing manipulators and the like.

Each node results in twenty or so macro definitions, `typedef`s, and functions. In particular, the node with name (and type) *`nname`* defines the `typedef` *`nname`* and the macros `N`*`nname`* to create a new instance of an *`nname`* node and `D`*`nname`* to delete an existing instance. The macros `I`*`nname`* and `F`*`nname`* give the name of procedures to call just after an *`nname`* node is created and just before it is deleted. Classes are not created directly, but result from type converting other classes or nodes using the macros described below.

A full set of macros are generated to manipulate type. Given a parent/child chain of length n, `idlc` generates $(n-1) \times n$ macros to convert a node of one type in the chain to another type in the chain. The `K`*`nname`* macro is the response returned by the `Typeof` macro when applied to an *`nname`* node. Applying `Typeof` to a class returns the `K` macro for the node from which the class came.

With the exception of ports, the implementation details for processes fall outside IDL's scope; `idlc` generates no code for processes. Each port results in a procedure to either read or write the structure, depending on the port direction. Each port procedure accepts a descriptor for the file that holds, or will hold, the structure.

The sequence of steps used to define and add IDL constructs to a program are shown in figure 1. The IDL source file `p.idl` defines the the ports used in the program. The port definitions refer to three data-structures, each defined in a separate IDL source file (`d1.idl`, `d2.idl`, and `d3.idl`).

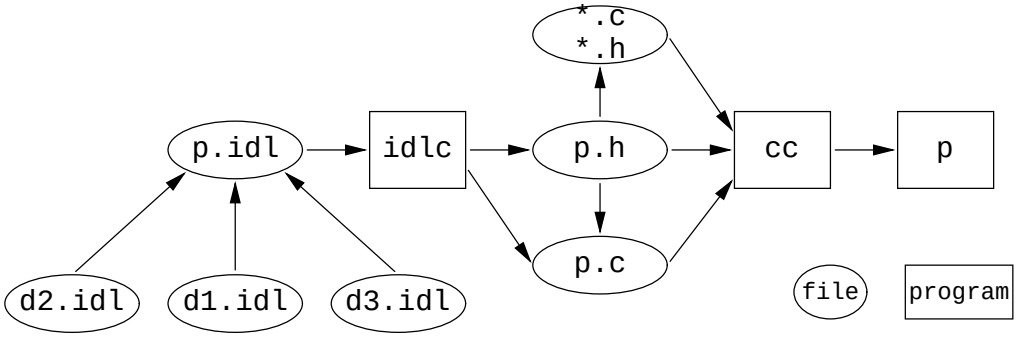

Figure 1: Adding IDL constructs to a program.

Port and structure definitions may be intermixed in a file; the `.idl` files shown in the figure are just one of many possibilites.

The IDL compiler `idlc` generates the C code for the ports (in `p.c`) and a set of data-structure macros (in `p.h`). Each port defined is implemented as a C function. The implementor uses the port functions and macros in the rest of the code needed by the program (`*.c`, `*.h`). The result after compilation and linking is the program (`p`).

The Work

This work is part of an effort studing how transformations can be applied to code to discover design decisions [ROL90]. The transformations "reverse compile" the code by replacing a section of code implementing some concept with a construct emphasizing what the concept does and de-emphasizing how the concept is implemented. The approach is similar to procedural abstraction, but the concept being abstracted doesn't have to be a procedure (for example, it could be "abbreviate a deeply nested field reference by assigning to a pointer") and doesn't have to be abstracted to a procedure (the previous example might be transformed to a Pascal-like `with` statement).

The task is to come up with a system for making such transformations. The expectation is to be able to explore possible transformations iteratively by interactively specifying them, applying them to code, and seeing what results. The immediate task is to determine how to transform code.

Transformations can be applied to the source code itself or to some representation of the source code. Despite the advantage of familiarity, using source code as the basis for transformations was rejected due to the complexity introduced by concrete syntax. Familarity can be maintained by moving to an abstract-syntax representation, raising the question of providing the intermediate representation.

IDL was chosen as the intermediate representation design and implementation tool for two reasons. The first is that this is the kind of task IDL was designed to handle, and we wanted to see how well it would handle it. The second reason is that other projects pursuing similar work have used IDL in similar situations [CD89].

Translating C to Abstract Syntax Trees

The two essential parts of translating C to an abstract syntax tree representation are the representation itself and the translation from C. Both parts hinge on finding an acceptable grammar for C, acceptability being measured by the directness with which the grammar describes C syntax. The grammar, suitably decorated with IDL-generating actions, performs the translation. The grammar also serves as a model for the representation with the nonterminals serving as candidates for the abstract-syntax tree nodes.

With grammar in hand, implementing the transformer is straight-forward, if not a little tedious. Figure 2 gives an example. Figure 2a shows the IDL node and class definitions for the storage specifiers prefixing a type declaration.[1] Storage specifiers are all more or less the same, so they have been abstracted into the single IDL class `Decl_spec_class` (Figure 2a, lines 10–13). Each storage specifier is defined as a node with no attributes (lines 16–18); as an optimization, the `For` directive on line 14 tells IDL to represent each storage specifier as an integer instead of an empty structure. Again, storage and type specifiers are more or less the same, so they have been abstracted into a `Decl_spec_class` (lines 6–8). Finally, a declaration specification `dcln_spec_node` for a type definition is a sequence of zero or more declaration specifiers (lines 3–4).

Figure 2b shows the part of the C grammar, adopted from [MTCY87], dealing with storage specifiers. The actions contain code that create the abstract syntax tree. Each storage specification token creates the associated IDL node using an `N` create macro and converts the node to the `Storage_spec_class` (lines 9–15 in figure 2b). The first storage specifier encountered creates a new declaration specifier node using `new_dsn()`, a local C routine (lines 1–3). Otherwise, the storage specifier is added to the sequence (lines 4–6). In either case the storage specifier is converted up to a declaration specifier class.

Figure 2 also illustrates the translation from grammar constructs to IDL structures. One-of choices, like the `storage_spec`, are unions and implemented as a class with each child representing a possible choice. Lists have a natural implementation as sequences. The other possibility, not shown in the figure, is a construct with a constant number of (perhaps optional) subconstructs; the `if` statement is an example where the subconstructs are the boolean expression and the true and false statement groups. These record-like constructs are naturally implemented by IDL nodes.

Abstract Syntax Tree Transformers

Given an IDL-decorated YACC grammar for C, a transformer for generating abstract syntax trees from C code follows immediately. A transformer working in the opposite direction, from an abstract syntax tree to C code, requires a bit more work. A code-generating transformer is a tree walker; each node outputs text and initiates walks of its children. A class represents a set of possible node types, and its appearance in the tree is taken as an appearance of one of the nodes. The walking code for a class is a case statement with one branch for each of its possible node types. Each branch converts the class to a node of the associated type and recursively walks the node.

Generating C from an abstract syntax tree requires code for every node and class in the IDL definition. With around 120 nodes and classes, an alternative to hand-coding is desirable. Fortunately,

[1] The prefix may also contain type specifiers like `int` or `volatile`. Type specifiers are handled in the same way as storage specifiers and have been omitted from the example.

```
1  -- The storage and type specifiers applicable to a data-type or
2  -- function declaration.

3     dcln_spec_node =>
4        specs          : Seq of Decl_spec_class;

5  -- A declaration specifier.

6     Decl_spec_class ::=
7         Storage_spec_class
8       | Type_spec_class;

9  -- The set of recognized storage specifiers.

10    Storage_spec_class ::=
11        auto_storage          | extern_storage
12      | register_storage      | static_storage
13      | typedef_storage;

14    For Storage_spec_class Use Representation Enumerated;

15 -- The storage specifiers.

16    auto_storage      => ;    extern_storage  => ;
17    register_storage  => ;    static_storage  => ;
18    typedef_storage   => ;
```

a) IDL definitions.

```
1  dcln_spec
2   : storage_spec
3       { $$ = new_dsn(Storage_spec_classToDecl_spec_class($1)); }

4   | dcln_spec storage_spec
5       { appendrearSEQDecl_spec_class($1->specs,
6           Storage_spec_classToDecl_spec_class($2)); }

7     /* stuff deleted */
8   ;

9  storage_spec
10  : auto_t
11      { $$ = auto_storageToStorage_spec_class(Nauto_storage); }

12  | typedef_t
13      { $$ = typedef_storageToStorage_spec_class(Ntypedef_storage); }

14    /* stuff deleted */
15  ;
```

b) IDL in a YACC grammar.

Figure 2: An Example IDL definition and its use.

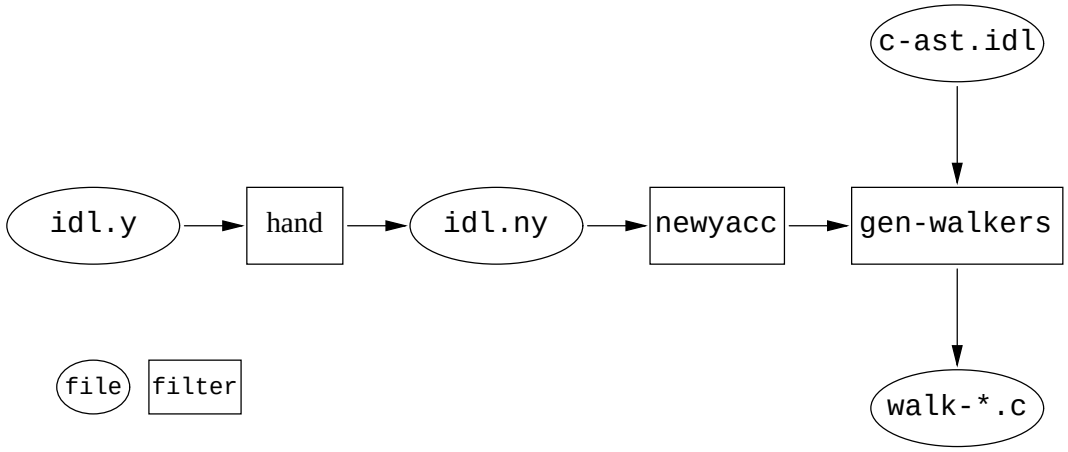

Figure 3: Automatically generating tree-walking code.

it is possible to automatically generate traversal code by using the YACC grammar for IDL description files and the `newyacc` parser generator [PC89]. The process is illustrated in figure 3. The YACC actions in the grammar are stripped out and NewYacc actions to generate the C code added; this is done by hand. The modified grammar is submitted to `newyacc`, resulting in a parser that accepts as input an IDL description of a structure and produces as output a set of C functions, one per node and class, that walk the nodes and classes defined by the input description.

The generated traversal routines are compiled and collected in a library. The routines only contain code to traverse the tree; any other functions they need to perform while walking the tree have to be added. The usual technique is to copy and modify the generated routines; the remaining, unmodified, routines are loaded from the library. Using this technique, the abstract syntax tree-to-C transformer modified around 85 (about $2/3$) of the generated routines.

A number of other transformers have also been produced using the library substitution technique. Of the two most interesting transformers, one derives a call-graph from an abstract syntax tree and the other modifies the abstract syntax tree to allow subroutine entries and exits to be tracked. Together the two produce a program-execution animation for the EDGE graph manipulation system [PT90].

Experience

Comments about experience with IDL are separated into two areas: general experience and work-related experience.

General Experience

We found we could pick up and use IDL in a couple of days. The SCORPION IDL compiler is reliable; we used only a few of the other tools (formatters mostly) less often, and also found them reliable. The generated code was acceptable; only one non-serious problem was uncovered during the course of the work. The documentation, both man pages and tutorials, is complete and usable.

The resources, in time and space, required by IDL, SCORPION, and the generated code seemed a little high, but this is a subjective guess. Resource requirements imposed no extra difficulties during the work or on the resulting system, but this may be an artifact of the filter-oriented approach to transformations in which the abstract syntax trees are transient, produced by one filter and consumed by the next. Measurements taken by applying the transformations to the Berkeley Magic CAD code suggest that, on average, the abstract syntax tree is seventeen times larger, when measured in bytes, than the code from which the tree was generated.

It is interesting to consider IDL from the viewpoint of object-oriented principles. This view is suggested by the presence of node-field inheritance along the subtyping chain. Being able to factor identical components into a common ancestor is an important and useful feature. However, difficulties arise from IDL being both too object oriented and not object oriented enough.

Implementing object-oriented features in a non-object-oriented language is one of the more difficult forms of programming even when the language being used provides relevant support for the features being implemented. In the C implementation of IDL, the programmer is responsible for maintaining the type hierarchy established by the IDL description; some indication of this responsibility can be seen in figure 2b. The significant structural differences between entities along the same type chain requires that type information be respected. C's limited approach to type compatibility requires that the programmer cover the difference between what C provides and IDL expects, and C's "poof - you're an `int`" style of type conversion requires extra diligence to avoid and detect mistakes.

With respect to not being object oriented enough, one frequently commented on characteristic of object-oriented systems is the ability to share code among related objects. IDL is able to directly associate code only for creating and deleting nodes, and, since nodes are at the bottom of the type chain, the code doesn't migrate from type to subtype. The latter problem was revealed when it became necessary to tag each abstract syntax tree node with the name of the source file from which it came. Adding the file-name field was simply a matter of modifying the root class for the abstract syntax tree. Making sure the file-name field was properly initialized, however, is a bit more tricky, potentially requiring a modification to every YACC grammar-rule action that creates a node. Fortunately, the `I` node-initialization macros could be used to insure proper initialization, and it was a relatively simple matter to use `newyacc` to read the `.idl` definition file and generate the set of `I` macros and associated procedures, one per node, to do the proper initialization. However, if the `I` macros had been otherwise used providing proper initialization would have been much more difficult.

IDL's inability to directly associate arbitrary code with a class or node is consistent with the clear separation in its computational model between transformation processes and the data on which they act. However, experiments with `newyacc` and `.idl` files containing specially tagged comments show that being able to directly associate code with classes and nodes is a convenient and useful feature. The experiments also revealed that associating several independent section of code, for different transformations for example, with the same IDL description is likely to be the central problem with this technique.

To summarize the comments on general experience, the ideas and motivation behind IDL are sound and important ones, and to have IDL available for use is a great help. Although one of IDL's strengths is that it can be mapped to almost any host language, most of its weaknesses result from instances where it is mapped to a host language not providing enough support for IDL features. Such

implementations lead to situations where the features are either ignored or underused, which is unsafe and wasteful, or are shored up with extra code, which is distracting and difficult.

Work Experience

At a basic level, the capabilities provided by IDL are a good match with the requirements of the work being done. The transformers need some kind of intermediate representation and IDL is good at creating internal representations. However, at other levels IDL proved to be less helpful to the point of requiring significant effort to achieve objectives. In particular, IDL tended to fall short when it came to providing representational mutability and structural linkages.

Representational mutability refers to the ability to change the basic structure of abstract syntax trees by adding new or modifying existing node types. Representational mutability should not be confused with changing a particular instance of an abstract syntax tree, which fixes the set of node types and modifies the graph constructed from nodes of the fixed set of node types. Representational mutability is required whenever the transformations move the internal representation beyond its defined structure, something which should occur often given the desire to explore transformations providing semantic abstractions.

IDL runs counter to representational mutability largely because of its heavy-weight redefinition cycle: stop, modify the structure, generate the code, compile the code, link the code, restart. Exploratory code transformations should not require a familiarity with the IDL redefinition cycle. Even if it were possible to hide the redefinition cycle behind the interface of a transformation system, the nature of the cycle, particularly with respect to the time taken, would ruin all but the simplest forms of interaction with the system.

A counter to the previous paragraph is the claim that IDL wasn't designed to provide interactive representational mutability and trying to use it as such is a misapplication. Although we agree with the claim, we argue against it on two points. The first is that some form of representational mutability, independent of response time, is becoming a feature in the kinds of systems with which IDL could be associated [Jor90]. The degree to which the association will be successful depends on how well IDL accommodates the mutability required. The second point in the argument against is that the "design, implement, evaluate" development cycle imposes representational mutability on any system as it evolves. The example of tagging nodes with file names described in the previous section is just one of the many such examples that occurred during this work. Since mutations (also known as "changes") are necessary and expected, IDL must insure the cost it extracts for making the changes is not too high.

The second higher-level requirement of this work is *structural linkage*, which is the ability to establish and maintain relations between components in the system being developed. Roughly speaking, structural linkage can be considered the prerequisite for easily building an all-encompassing makefile for the system. There are at least two motivations behind structural linkages. The first is the usual software engineering motivation of having a simple and correct method for building a consistent version of the system. The second motivation comes from the observation that almost all the information needed for this work comes from the grammar for C. Ideally, it should be possible to derive all the subsystems of the final system from the grammar. The second motivation for structural linkages is concerned with getting as close to the ideal as possible.

With respect to the first motivation for structural linkage, IDL caused problems in two areas. The first is sharing common information. Each transformation filter uses the same IDL structure definition for the abstract syntax tree, but uses different sets of ports to input and output the tree. IDL bases its file generation on the port set definition, leaving each filter with a set of generated files that differs from the files generated for other filters in filename only. This problem can (and has been) alleviated somewhat by noting that each filter's port set belongs to one of three classes: input only (for a filter at the start of a chain), input and output (for filters in the middle of a chain), and output only (for a filter at the end of a chain). This reduces, but doesn't eliminate, repetition.

The second problem also arose because IDL generated code with respect to ports. The generic tree walking routines described in an earlier section needed the abstract syntax tree structure definitions generated by IDL before they could be compiled and loaded into a library. However, since the routines are generic, that is, created without reference to any particular filter, there were no files with the needed definitions. The solution—use the necessary files defined for a filter—worked and was simple, but is unsatisfactory because it complicates matters more than a proper solution and it establishes an otherwise unnecessary dependency between generic routines and a particular filter.

Conclusions

This paper has described the IDL intermediate description language tool suite, described its use in work on developing a system for transforming programming language source code, and evaluated its contributions to the work. IDL is a useful to have at hand when dealing with intermediate representations for source code. Informal cost/benefit analysis suggests that the IDL implementation provided by SCORPION has a fixed benefit for a large (in terms of time and space) cost, and so is not a lightweight tool.

The problems identified with IDL can be grouped into two classes. The first class results from mapping IDL into a language providing inadequate support. An implementation of IDL in a language that has, for example, a strong and flexible type system and garbage collection would eliminate most of these problems. The second class of problems result from requirements in software development technologies that over-extend or run contrary to the assumptions made by IDL. Two examples taken from the work reported here are the need to make dynamic and expedient changes to the internal representation and the desire to directly associate code with various parts of the intermediate representation. Problems from the second class are probably going to be harder to solve since they deal with fundamental IDL issues.

References

[CD89] G. Cabral and H. E. Dunsmore. A design format transformation exercise. Technical Report SERC–TR–35–P, Purdue University, Software Engineering Research Center, February 1989.

[Jor90] M. Jordan. An extensible programming environment for Modula-3. In *Forth ACM SIGSOFT Symposium on Software Development Environments*, pages 66–76, December 1990.

[MTCY87] William M. McKeeman, Susan Trager, Joshua L. Cohen, and Ting Yang. C grammars. Technical Report TR–87–02, School of Information Technology, Wang Institute of Graduate Studies, February 1987.

[PC89] James M. Purtilo and John R. Callahan. Parse tree annotations. *Communications of the ACM*, 32:1467–1477, December 1989.

[PT90] Frances Newbery Paulisch and Walter F. Tichy. EDGE: An extendible graph editor. *Software—Practice And Expierence*, 20(S1):63–88, June 1990.

[ROL90] Spencer Rugaber, Stephen B. Ornburn, and Richard J. LeBlanc, Jr. Recognizing design decisions in programs. *IEEE Software*, 7(1):46–54, January 1990.

[Sno89] Richard Snodgrass. *The Interface Description Language: Definition and Use*. Computer Science Press, 1989.

[Sno91] Richard Snodgrass. SCORPION system tutorial. Technical Report Rel. 5.0, University of Arizona, Tucson, Az., March 1991.