AD-A181 937 RELATIONSHIP BETWEEN IDL (INTERFACE DESCRIPTION LANGUAGE) AND STRUCTURE E (U) CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST P H FEILER SEP 86 CMU/SEI-86-TM-13 ESD-TR-86-228 F/G 12/5

1/1

UNCLASSIFIED

NL

MICROCOPY RESOLUTION TEST CHART

ESD-TR-86-221

2

Technical Memorandum
SEI-86-TM-13

Carnegie-Mellon University
Software Engineering Institute

Relationship between IDL and Structure Editor Generation Technology

by

Peter H. Feiler

September 1986

ADA181937

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNLIMITED, UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT UNCLASSIFIED, UNLIMITED, DTIC, NTIS
4. PERFORMING ORGANIZATION REPORT NUMBER(S) SEI-86-TM-13	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-86-220
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST. 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAMMING OFFICE
6c. ADDRESS (City, State and ZIP Code) CARENGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE HANSCOM, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JPO 8b. OFFICE SYMBOL (If applicable) ESD/XRS1	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628 85 ~~A~~ C 0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: 63752F PROJECT NO.: N/A TASK NO.: N/A WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) RELATIONSHIP BETWEEN IDL AND STRUCTURE EDITOR GENERATION TECHNOLOGY	
12. PERSONAL AUTHOR(S) PETER FEILER	
13a. TYPE OF REPORT FINAL 13b. TIME COVERED FROM ... TO ...	14. DATE OF REPORT (Yr., Mo., Day) SEPTEMBER 86 15. PAGE COUNT 12
16. SUPPLEMENTARY NOTATION N/A	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

THIS PAPER DISCUSSES OBSERVED COMMONALITIES AND DIFFERENCES BETWEEN IDL AND STRUCTURE EDITOR GENERATION TECHNOLOGIES. IDL (INTERFACE DESCRIPTION LANGUAGE) IS TECHNOLOGY FOR GENERATION OF TOOL INTERCOMMUNICATION SUPPORT WITH ROOTS IN COMPILER GENERATION. STRUCTURE EDITOR GENERATION TECHNOLOGY HAS ITS ROOTS IN SYNTAX-DIRECTED EDITORS. IT PRODUCES ENVIRONMENTS FOR INTERACTIVE VIEWING AND MANIPULATION OF FORMALLY SPECIFIED STRUCTURES. BOTH TECHNOLOGIES USE A FORMAL NOTATION FOR STRUCTURAL AND CONSTRAINT DESCRIPTIONS. FROM THESE DESCRIPTONS BOTH GENERATION TOOLS AUTOMATICALLY PRODUCE SOFTWARE FOR READING, WRITING, AND MANIPULATING INSTANCES OF THE DESCRIBED STRUCTURES, AS WELL AS FOR CHECKING SPECIFIED CONSTRAINTS ON INFORMATION CONTAINED IN THE STRUCTURES. THE IDL TECHNOLOGY EMPAHSIZES GENERATION OF BATCH-ORIENTED APPLICATIONS WHILE THE STRUCTURE EDITOR GENERATION TECHNOLOGY IS TAILORED TO SUPPORTING INTERACTIVE APPLICATIONS. STRUCTURE EDITOR GENERATION TECHNOLOGY HAS BEEN APPLIED TO ITSELF, I.E., TO BUILDING AN INTERACTIVE STRUCTURE EDITOR GENERATION ENVIRONMENT.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED [✓] SAME AS RPT [✓] DTIC USERS [✓]	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED, DTIC, NTIS
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL H. SHINGLER	22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630 22c. OFFICE SYMBOL SEI JPO

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE

Technical Memorandum
SEI-86-TM-13
September 1986

Relationship between IDL and Structure Editor Generation Technology

by

Peter H. Feiler
Software Engineering Institute

This work was sponsored by the Department of Defense.

Table of Contents

List of Figures

Relationship between IDL and Structure Editor Generation Technology

Peter H. Feiler

Abstract. This paper discusses observed commonalities and differences between IDL and structure editor generation technologies. IDL (Interface Description Language) is technology for generation of tool intercommunication support with roots in compiler generation. Structure editor generation technology has its roots in syntax-directed editors. It produces environments for interactive viewing and manipulation of formally specified structures. Both technologies use a formal notation for structural and constraint descriptions. From these descriptions both generation tools automatically produce software for reading, writing, and manipulating instances of the described structures, as well as for checking specified constraints on information contained in the structures. The IDL technology emphasizes generation of batch-oriented applications while the structure editor generation technology is tailored to supporting interactive applications. Structure editor generation technology has been applied to itself, i.e., to building an interactive structure editor generation environment.

1 Introduction

The intent of this article is to share with the reader some observations about the relationship of IDL (Interface Description Language) technology and structure editor generation technology. The origins of the two technologies vary, and each of them addresses a different problem domain. IDL technology has its roots in compiler technology and provides support for tool intercommunication. Structure editor generation technology provides a shell for interactive applications that manipulate structures. In both cases, the construction of applications that manipulate structures is supported through a generation approach. The characteristics of the structures are described in a high-level notation. Both technologies have chosen similar notations for the descriptions. From such a description code is generated for inclusion in an application. The differences in the application domains are reflected in the functionality of the code being supplied to the application.

For the discussion in this article we have chosen the Dose structure editor generation environment and its Representation Description Language RDL [10] as a representative of structure editor generation technology. We first give a short history of both technologies. Then, the architectures implied for the applications are compared. This is followed by a discussion of the structural description facilities, the support for composing and viewing structures, and the notation for expressing constraints on structures as found in IDL and in RDL. The article concludes with a description of the two generation environments.

2 History

The IDL technology effort originated in compiler technology and grew out of the PQCC project at Carnegie-Mellon University. One component of this technology is a language (referred to in this article as *IDL*) that was developed as a generalized data definition language to permit interfacing of results from different project components. IDL was used to define the intermediate representation for Ada™ called Diana [3]. A full description of IDL can be found in [13]. The other component of IDL technology is a generation tool. Generation of software based on an IDL description and the tool supporting the generation are discussed in a thesis by Lamb [12]. More recently, the SoftLab project at the University of North Carolina has been building a Unix™/C-based IDL generation tool [17].

Structure editor generation technology has its roots in syntax-directed editors. An early syntax-directed editor system, Emily [9], based its model of structure manipulation on BNF. Later the Mentor system [4] started to use as its structural representation an abstract syntax tree representation — as was done in Diana. In the Aloe system [6] of the Gandalf project and later other structure editor systems [10, 16], structural information was not handcoded into the system, but described in a notation with many similarities to IDL. From that description an implementation was generated. Structure editors provide a good basis for interactive applications dealing with structures. Originally, syntax-directed editors supported incremental program construction, i.e., programming in the small. Later structure editor-based environments such as Gandalf, a complete development environment [15], structure editor generation environments, e.g., AloeGen [5] and Dose [8], and non-programming applications [14, 7] were built.

3 Architecture For Applications

IDL technology assumes that an application reads in one or several structures, manipulates them, and then writes them out. For that purpose the IDL generation tool produces from an IDL description reader and writer routines, structure manipulation routines, runtime support routines (e.g., specialized memory allocation routines), and constraint checking code. Reader and writer routines operate both with an external ascii representation to aid portability across machine architectures, and with a binary representation for efficiency. Structures that are read or written can be subsets of structures maintained within the application, i.e., the reader and writer act as filters. An application is built on top of these routines. Constraint checking code is available as a separate program that can be invoked on structures that are read or written by applications. An application can be composed of one or more *IDL processes*. An IDL process represents a logical processing unit that takes structures as input, manipulates them, and produces structures as output. IDL processes are interconnected through their input and output structures. Several IDL processes can be mapped into one application program. In this case reader and writer routines act as filters for in-core structures. The left diagram of Figure 1 illustrates such an IDL-based application architecture.

Structure editor generation technology supports the construction of application for interactive manipulation of structures. Such applications are created from an interactive structure manipulation shell that is provided by the generation system. The shell consists of two layers. This is illustrated for the Dose environment in the right diagram of Figure 1, yet applies to other structure editor systems as well. The lower layer of the shell corresponds to the generated IDL routines, i.e., it consists of reader and writer routines, structure manipulation routines, and runtime routines

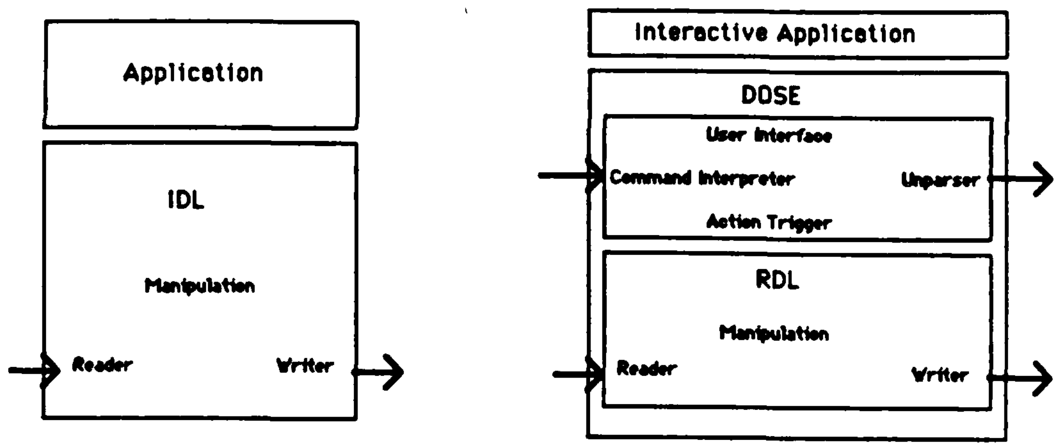


Figure 1: Comparison of IDL and RDL Application Architectures

such as specialized memory management. Readers and writers work according to views, i.e., they can read and write substructures and as a result act as filters for passing structures between different instances of structure editor shells. This layer of the shell is complemented with a second layer that provides facilities for interactive manipulation of the structures and for triggering constraint checking and application-specific processing. The interactive facilities include a window- and menu-based user interface, a command interpreter with on-line help support, and capabilities for viewing, browsing, and modifying structures. Multiple views, both textual and graphical [7], are supported. Display of views is generated at runtime on demand, i.e., only for the part of the structure the user actually desires to examine. Views permit the user as well as application code to interact with a subset of the structure. Multiple views providing access to different levels of detail in the structure effectively provide browsing facilities. The trigger facility of the shell is activated whenever the user manipulates a part of the structure. A symbol processing mechanism is invoked when an identifier is entered, touched, or removed in the structure. Constraint checking routines and application processing routines are invoked both on movement of the editor cursor and on manipulation operations. Triggering at small granularity permits incremental checking and processing, resulting in an interactive application.

4 Structural Description

The structural description specifies structures in terms of typed, attributed, directed graphs. Productions define node types, and the names and types of their attributes.

IDL suppc ':; the following primitive types: boolean, integer, rational, and string. Other types are defined through composition. IDL supports sets, sequences, user supplied type declarations, and classes. Sets and sequences can be applied to any attribute. They can be empty. User defined type declarations refer to types that are implemented in separate packages. This permits new types to be introduced and different implementations for existing types to be given, e.g., representation of an integer with range 1..100 as a byte. Classes define unions of node types. IDL supports a non-hierarchical class structure. Enumerations can be expressed as classes of productions with no attributes for the productions.

In RDL the primitive types are integer, string, and identifier. In the latter case definition and use of identifiers are distinguished. This simplifies the mechanisms for symbol processing. Composition facilities include enumerated types, class, sequence, and optionality. Different from IDL, enumeration is represented explicitly, permitting the generation system to optimize its representation. Boolean is defined as an enumerated type. In RDL a class definition consists of a list of production names. Sequences apply to any attribute and are expected to have one or more elements. Optionality indicates whether an attribute is required or not. This results in cardinality restrictions on attribute values (attribute -> 1; optional attribute -> 0 or 1; sequence -> 1 or more; optional sequence -> 0 or more). Sets were not felt to be necessary. Instead, the notion of symbols (identifiers) and associated name and symbol tables are explicit in structure editors.

5 Structure Composition and Views

IDL and structure editor generation technology take different approaches to dealing with structure composition and views on structures. IDL takes a process-oriented view of the application whereas structure editor generators take a data base/object base-oriented view.

In IDL, structures are specified with *structure declarations*. Structure declarations consist of a set of productions and classes, and a reference to the type of the structure root. Structures can be refined or derived from other structures. Refinement allows the specification of a more detailed representation of the old structure. Derivation permits the specification of a new structure by indication of differences with another structure — a convenience for dealing with several similar structures. An IDL process is composed of input and output structures as well as structures that are maintained within an application.

In RDL, a structure is specified in a *representation description (RD)* through a set of productions and classes, and a structure root. This structure is decomposed into substructures. Substructures are specified by defining views on the structure. A view specification is attached to each production. It indicates a subset of the node types and attributes that are accessible. They can be accessible with modification rights or read-only. Views are augmented with templates for generation of displayable representations. The same template information can be used to generate parsers. A set of views showing a structure at different levels of detail provide the basis of a structure browsing facility. Conditional views allow queries to be specified on structures. Examples of queries that can be defined through views are "what are all the procedures" or "which modules still contain errors." A more extensive discussion of the power of structure editors as structure viewers can be found in [7].

In RDL, an application can make use of multiple representation descriptions. The specification of an attribute in one RD can refer to a class in another RD. This permits partitioning of representation descriptions, which is desirable for applications with large structures such as development environments (see for example the Gandalf system [15]). This partitioning of the RD implies that structures based on these descriptions are managed accordingly. Each structure partition can be stored to and loaded from secondary storage as a separate unit. Multiple representation descriptions are managed by the generation environment DOSE rather than through additional language constructs.

6 Constraints and Processing

IDL supports the specification of assertions on structures. The assertion language has constructs for conveniently dealing with graph structures, sets, and sequences. Assertions are expected to hold when a structure enters and leaves an IDL process. No constraints are implied while the process is manipulating the structure. In the SoftLab implementation assertions are translated into a checker program that can be invoked on a structure it has written or read before. If not translated into checking code assertions still act as specifications in that they document constraints on the structure that the application is expected to adhere to at certain times.

Structure editors emphasize interactive manipulation of structures and incremental processing. For that purpose two triggering mechanisms are provided with structure editors: an *action routine* mechanism, and a *symbol processing* mechanism.

The action routine mechanism activates actions as the structure is manipulated or traversed, e.g., by editing or cursor motion commands. Constraints and application processing are associated with node types, and are activated when a node of that particular type is operated on. Constraints and application processing are expressed in a high-level procedural language with support for traversal and manipulation of graph structures and sequences. It supports the description of both checking and processing. Static semantic checking as well as additions and changes to the structure, for example derivation of a procedure call template with actual parameters from its specification, can be expressed. In order to fully support application code an escape into an underlying programming language is provided where necessary. Efforts on other structure editor generation systems have resulted in incremental attribute grammars [2], and in active constraints [11] as constraint languages.

The symbol processing mechanism activated routines for handling definitions and uses of identifiers. It is triggered by operations on instances of the primitive types identifier-definition and identifier-use. The generation environment supplies a default implementation for symbol processing, which can be augmented by the user of the generation environment. The user does so by supplying a description of the new symbol table structure using the structure description language as well as of the necessary processing in terms of the graph-oriented processing language.

7 Generation Environment

The generation environment for IDL consists of your favorite text editor for developing IDL descriptions, an IDL translator that processes IDL descriptions, an IDL library that contains structure description independent IDL routines and is linked in with the generated routines and the application code, and some debugging tools. Compilation and linking is done with application language tools. The SoftLab implementation of the IDL translator maps structural information directly into the application language, taking advantage of its structural constructs and type mechanisms. Lamb's translator chose a table-driven implementation for the generated code. It has been realized that changes in the structure can require massive recompilation of the application [1]. Therefore, both an efficient production implementation and a more flexible development implementation have been proposed. Two debugging tools are provided with the SoftLab implementation. One program graphically plots externally stored IDL structures. A second program generates cross-reference information for structures.

The most obvious generation environment for structure editors is similar to that of IDL. On several occasions, however, the generation environment has been built as an instance of a structure editor based interactive application. Both AloeGen and Dose are good examples. The formal notation of structure descriptions is described in itself resulting in an instance of a structure editor shell for structure descriptions. These interactive generation environments take full advantage of the capabilities provided by the structure editor shell. Users can structurally manipulate descriptions. Users can develop the structural description, the constraints and application actions, and auxiliary structures in the same environment. Multiple views provide browsing and querying facilities, reducing the amount of information seen at any one time. Through these views the user can examine structures and cross-reference information interactively. The description is checked for constraint violation interactively as the description is being developed (i.e., the description of the formal notation includes constraints and processing).

AloeGen takes a table-driven approach for the implementation. The generated system consists of a structure independent part and a structure dependent part. The structure independent part is a shell or kernel consisting of the manipulation routines, reader and writer, user interface, and command interpreter. The structure dependent part consists of a table containing the structural information, and a collection of routines representing the constraints and the application actions. A new instance of a structure editor application is created by linking the kernel code with the constraint and application routines and linking in or dynamically reading in the table. AloeGen accomplishes the translation of a structural description or a constraint description through multiple textual views. One view of the description is that of the description notation or constraint language, whereas a second view transforms the description into C code representing the table and the checking and processing routines, i.e., the structure editor is not only a tool for developing descriptions, but also acts as the generation tool.

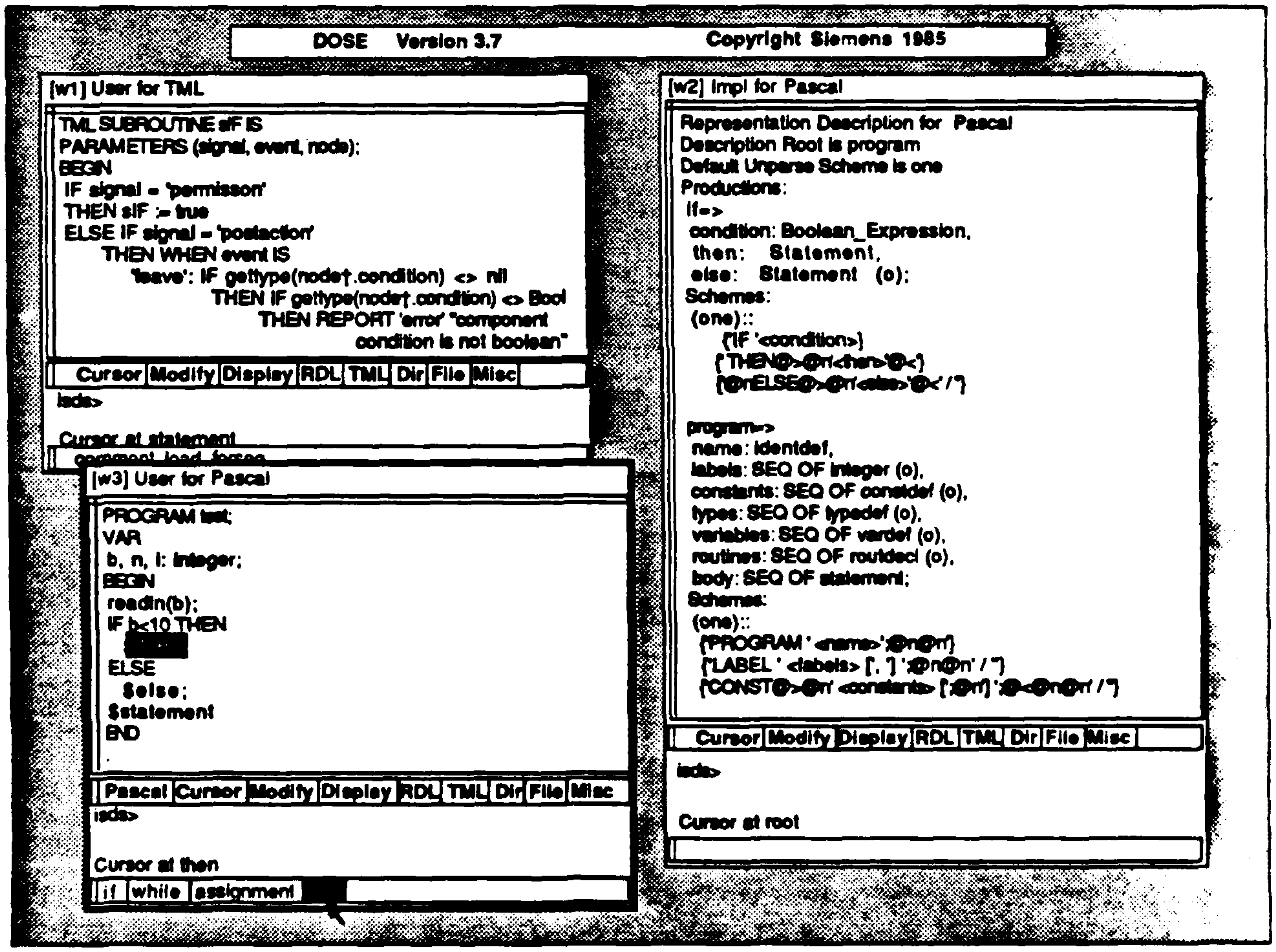
Dose takes a different approach. Structural information and constraint descriptions are created by the structure editor as a structure. This structure represents the structure dependent part of the system. This structure is directly interpreted by the structure editor kernel, i.e., the Dose structure editor system does not require a translation or generation step. The structure representing a structure description or constraint description is read into the editor and accessed using the same kernel facilities that are used to deal with the application structure. The structure editor does so by interpreting a description for structure descriptions, which is represented as a structure in turn. Fig. 2 gives a structural description of an RDL production described in RDL.

```
production =>
      prodname : identdef,
      arity : integer,
      components : SEQ OF component,
      schemes : SEQ OF scheme,
      impl : impl_info(o);

   schemes:
      (structure_view)::
         {<prodname*> ' => @>@>@n'
          <components*> [',@n'] ';@<'}

      (full_view)::
         ....
```

Figure 2: Description of a Production in RDL



```
TML SUBROUTINE sIF IS
PARAMETERS (signal, event, node);
BEGIN
 IF signal = 'permisson'
 THEN sIF := true
 ELSE IF signal = 'postaction'
      THEN WHEN event IS
          'leave': IF gettype(node↑.condition) <> nil
                        THEN IF gettype(node↑.condition) <> Bool
                             THEN REPORT 'error' "component
                                          condition is not boolean"
```

```
isds>

Cursor at statement
  comment load forsee
```

```
PROGRAM test;
VAR
 b, n, i: integer;
BEGIN
 readln(b);
 IF b<10 THEN

 ELSE
   $else;
 $statement
 END
.
```

```
isds>

Cursor at then
```

```
Representation Description for  Pascal
Description Root is program
Default Unparse Scheme is one
Productions:
 If=>
  condition: Boolean_Expression,
  then:   Statement,
  else:  Statement  (o);
 Schemes:
  (one)::
      {'IF '<condition>}
     {' THEN@>@n'<then>'@<}
     {'@nELSE@>@n'<else>'@<' / '}

 program=>
  name: identdef,
  labels: SEQ OF integer (o),
  constants: SEQ OF constdef (o),
  types: SEQ OF typedef (o),
  variables: SEQ OF vardef (o),
  routines: SEQ OF routdecl (o),
  body: SEQ OF statement;
 Schemes:
  (one)::
   {'PROGRAM ' <name>';@n@n'}
   {'LABEL ' <labels> [', '] ';@n@n' / '}
   {'CONST@>@n' <constants> [';@n'] ';@<@n@n' / '}
```

```
isds>

Cursor at root
```

Figure 3: Screen View of Prototyping Environment

The direct interpretation of structure descriptions turns Dose into a prototyping environment for description and language development. The user of the Dose generation environment can interactively switch between working on the description and testing it by creating structures based on the description. A screen snapshot of the Dose generation environment, showing a window with a Pascal structure description, a window with an application processing routine, and a window displaying an instance of the discribed structure, i.e., a Pascal program, is provided in Figure 3.

8 Conclusions

The comparison of IDL and structure editor generation technology are summarized in Fig. 4.

	IDL	RDL/Dose
Application Architecture	structure manipulation and read/write primitives	shell for manipulation by interactive applications
Primitive Types	boolean, integer, string, rational	integer, string, identifier
Composition	set, sequence, class	sequence, class, enumeration, optimality
Constraints	graph oriented assertion language	graph oriented assertion and processing language with symbol processing
Generated Code	direct encoding, table-driven	table-driven, interpretive
Generation Tools/Environments	collection of offline-oriented processing tools	structure-editor-based interactive environment

Figure 4: Summary of Comparison of IDL and Structure Editors

IDL technology and structure editor generation technology have their separate origins and were developed to address different application domains. They both have addressed the issue of more effectively building software that operates on structures (typed object bases). Formal high-level descriptions specify the structure and constraints. These descriptions can be translated into a variety of (quite efficient) implementations. The differences come in because IDL was targeted to support tool intercommunication, whereas with structure editors the emphasis was in providing a shell for building highly interactive applications.

Both technologies can profit from each other. IDL can benefit from an interactive generation environment based on a structure editor. It can also incorporate into its generation process different flexible implementation techniques and additional support routines, especially in supporting the development of interactive applications. Structure editor generation technology can take advantage of the implementations generated by IDL tools for the implementation of structure editors. The result could be very time and space efficient structure editor implementations.

References

[1] Lori A. Clarke, Jack C. Wileden, Alexander L. Wolf.
Graphite: A Meta-Tool For Ada Environment Development.
In *Proceedings of Second International Conference on Ada Applications and Environments*. IEEE Computer Society, April, 1986.

[2] Alan Demers, Thomas Reps and Tim Teitelbaum.
Incremental Evaluation for Attribute Grammars with Applications to Syntax-directed Editors.
In *Proceedings of 8th POPL Conference*. January, 1981.

[3] *Diana Reference Manual*
Carnegie-Mellon University, Department of Computer Science, 1981.

[4] Veronique Donzeau-Gouge, Gerard Huet, Gilles Kahn, and Bernard Lang.
Programming Environments Based on Structured Editors: The Mentor Experience.
Interactive Programming Environments.
McGraw-Hill Book Co., New York, NY, 1984.

[5] Robert J. Ellison and Barbara J. Staudt.
The Evolution of the GANDALF System.
The Joumal of Systems and Software 5(2):107-119, May, 1985.

[6] Peter H. Feiler and Raul Medina-Mora.
An Incremental Programming Environment.
IEEE Transactions on Software Engineering SE-7(5), September, 1981.

[7] Peter H. Feiler and Gail E. Kaiser.
Display-Oriented Structure Manipulation in a Multi-Purpose System.
In *Proceedings of IEEE COMPSAC 83*, pages 40-48. November, 1983.

[8] Feiler, P.H., Jalili, F., Schlichter, J.H.
An Interactive Prototyping Environment for Language Design.
In *Proceedings of Hawaii International Conference on System Sciences*. IEEE, January, 1986.

[9] Wilfred J. Hansen.
User Engineering Principles for Interactive Systems.
In *Fall Joint Computer Conference Proceedings*. 1971.

[10] Gail E. Kaiser and Peter H. Feiler.
Generation of Language-Oriented Editors.
In *Programmierumgebungen und Compiler*, pages 31-45. B. G. Teubner, Stuttgart, April, 1984.

[11] Gail E. Kaiser.
Semantics of Structure Editing Environments.
PhD thesis, Carnegie-Mellon University, May, 1985.
Technical Report CMU-CS-85-131.

[12] David A. Lamb.
Sharing Intermediate Representations: The Interface Description Language.
PhD thesis, Carnegie-Mellon University, May, 1983.

[13] John R. Nestor, William A. Wulf and David A. Lamb.
IDL - Interface Description Language: Formal Description.
Technical Report, Software Engineering Institute, Pittsburgh, PA, February, 1986.

[14] David S. Notkin.
Interactive Structure-Oriented Computing.
PhD thesis, Carnegie-Mellon University, February, 1984.

[15] David Notkin.
The GANDALF Project.
The Journal of Systems and Software 5(2):91-105, May, 1985.

[16] Thomas Reps and Tim Teitelbaum.
The Synthesizer Generator.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* Pittsburgh, PA, April, 1984.

[17] W. B. Warren, J. Kickenson, and R. Snodgrass.
A Tutorial Introduction to Using IDL.
Technical Report, Computer Science Department, University of North Carolina, Nov, 1985.
SoftLab Document No. 1.