Direct Implementation of Abstract Data Types from Abstract Specifications

BOUMEDIENE BELKHOUCHE, MEMBER, IEEE, AND JOSEPH E. URBAN, MEMBER, IEEE

Abstract—The development of correct specifications is a critical task in the software development process. This paper describes an alternative approach for the development of specifications. The approach relies on a specification language for abstract data types and a synthesis system. The system is capable of translating an abstract data type specification into an executable program. This process defines an alternative methodology that provides the necessary tools for the early testing of the specifications and for the development of prototypes and implementation models.

Index Terms—Abstract data types, abstract model, implementation models, language translation, prototyping, specifications, specification testing, synthesis, transformation rules.

I. Introduction

A CRITICAL phase of the software life cycle is the specification phase. The objective of the specification phase is to provide a formal description of the functionality of the system to be developed, and to serve as a basis for communication, design, testing, and verification of the software product. The significance of software specification is analogous to that of blueprints used in other engineering disciplines. However, unlike other engineers, software engineers have not yet fully adopted the use of specifications on a large scale. This attitude is partly due to the lack of software aids for developing specifications. Indeed, the production of accurate specifications is a complex task that requires automated tools. These tools include testing and debugging facilities to validate the specifications before the design process is started. As reported by Boehm [1], over 60 percent of the errors uncovered in several operational software systems were due to shortcomings in the specifications themselves. This high percentage of errors contributes significantly to the unreliability and prohibitive cost of software. It is therefore desirable that support tools be available to help software developers define error-free specifications.

Several tools and methodologies have been proposed to minimize software development problems. The concepts of stepwise refinement and hierarchical decomposition attempt to provide the software engineer with guidelines on how to solve a difficult problem through the use of abstraction levels and successive refinement stages. These manual methodologies emphasize a problem-solving discipline, but do not prevent the introduction of errors into the resulting product. Program testing and validation are required thereafter to assess the equivalence between the developed system and its specification. This validation process is arduous and needs to be performed every time a new software product is developed or modified. Consequently, even though these techniques help reorganize and master the complexity of the problem, they do not constitute a safeguard against the injection of new errors in the intermediate designs. The synthesis system described in this paper is intended to alleviate these problems. The automated system is a formalization of the stepwise refinement and hierarchical decomposition processes. Such a formal methodology guarantees the correctness of the programs generated by the system. That is, given that the transformation rules that translate a specification into an implementation are proved correct, any product generated by the synthesis process is also correct. Proving the correctness of the transformation rules is a one-time activity.

Issues associated with the specification phase were addressed in the research reported in this paper through the definition of a specification language and a processor for the language [2]–[4]. This processor is capable of automatically generating implementations of abstract data types from the associated high-level specifications. Several benefits accrue from the integration of this tool within an existing software environment. Such a system provides the necessary support for testing the specifications, and thus can be used as a validation tool. The system also allows the software developer to experiment with, and study different prototyes and implementation models that are automatically synthesized. Currently, the synthesis system is written in PL/I on a Honeywell 68/80 running under MULTICS. The input to the synthesizer is an abstract data type specification. The specification is analyzed, and if legal, PL/I or Pascal code that implements the specification is generated. The implementation and the necessary interfaces are then available for use by other programs. The early availability of implementations constitutes a definite advantage for error detection and precise specification.

The innovations that this approach introduces in software engineering are:

1) the human effort is mainly devoted to the requirements and specification phases;

Manuscript received May 31, 1984; revised July 31, 1985.
B. Belkhouche is with the Department of Computer Science, Tulane University, New Orleans, LA 70118.
J. E. Urban is with the Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA 70504.
IEEE Log Number 8607940.

2) the automatically generated prototypes are equivalent to the specifications;

3) the tesing of the specifications is greatly facilitated by the early availability of the generated prototypes;

4) the early feedback from the user can be obtained through observation of the behavior of the prototypes; and

5) the software life cycle defined by the synthesis system is shorter and less error-prone than the traditional software life cycle.

II. The Specification Language

The language design was influenced by ALPHARD [5], [6] and Berzins' language [7]. The reason for not adopting either language is because neither one provided the features necessary for automated support. The ALPHARD specification language was designed to provide a basis for manual program verification with no provisions for machine processing. Berzins' specification language was partially designed to illustrate the abstract model approach to specifications of abstract data types with an emphasis on human readability rather than machine processing.

The language model described here provides four basic abstract data structures and associated operations. These mathematical structures are: set, sequence, Cartesian product, and discriminated union [8]. To define an abstract data type, the specification developer chooses an abstract representation and defines the semantics of the abstract data type in terms of the operations available on the abstract representation. In contrast to the algebraic approach [9]–[11], this model is considered to be constructive and more amenable to proofs of correctness [12]–[14]. This advantage is due to the relationship between the abstract model approach and programming language concepts which allows a straightforward formulation of the mapping function from the concrete to the abstract object [15].

A. General Structure of a Type Specification

The specification of an abstract data type in the language describes the syntactic structure, and defines the functionality (i.e., semantics) of the abstract data type operations. Once defined, an abstract data type specification is viewed differently according to the nature of its usage. We identify three kinds of usage: 1) the using programs; 2) the synthesis system; and 3) the program verifier. Programs using the abstract data type need to interface with the specification. These programs must have access to the syntax part and knowledge of the semantics of each operation. The synthesis system uses the specification as input, determines its legality, and implements it. The program verifier needs both the specification and corresponding implementation to demonstrate their equivalence. In all aspects, the specification is used as an unambiguous communication vehicle among users of varied needs and requirements. The specification language must thus have sufficient expressive power to capture a type specification in a form that is suitable for all users.

TYPE <name, generic parameters>
INTERFACE <syntax of operations>
LET <abstract representation>
INITIALLY <initialization>
OPERATIONS <semantics of operations>
RESTRICTIONS <list of restrictions>
END <name>

Fig. 1. General structure of a type specification.

The unit of the specification language is the type specification. Fig. 1 shows the general structure of a type specification.

As can be seen from Fig. 1, a type specification consists of seven sections which are as follows.

1) Type Header Section: This section defines the abstract data type name that will serve as the type identifier to be used for type declaration. Generic parameters, if any, are associated with the type identifier. These parameters extend the definition of the type to a type schema. Language restrictions imposed on generic parameters are: 1) generic parameters must be simple identifiers; 2) generic parameters must be of language-defined type; 3) the operations available on generic parameters are automatically inherited; and 4) the values of generic parameters must be known at compile time. Restrictions on the generic parameters are stated in the **where** clause following the type header. These restrictions are global to the specification, and must be enforced for all the operations. This global view makes the restrictions behave as invariants [6]. Each restriction is evaluated statically. The results of the evaluation must be a true value, otherwise an exception is signaled. Restrictions that may be expressed in the language are of three types. These restrictions are: 1) a range restriction which constrains the range of values a generic parameter can assume; 2) a type restriction which constrains a generic parameter of type **type** to certain types; and 3) a property restriction which provides a list of properties of a generic parameter.

2) Interface Section: This section is the syntactic specification of the abstract data type. The types and the operations that may be exported to other specifications or programs are listed in this section. The information provided by the interface section is made visible outside the type specification to allow for proper usage of the type, and to enforce statically the legal interface between using programs and implementing programs. The available information consists of a list of type names and a list of operation headers. Each header describes the syntax of the operation which defines the name of the operation, and the type and position of each of the operation parameters. Imported types are listed in the imports subsection.

3) Abstract Representation Section: This section is used to define an abstract type structure in terms of user- or language-defined types by using structuring concepts

supported by the specification language. These structuring concepts are called abstract data structures and consist of sequences, sets, tuples, and discriminated unions. The primitive structures furnish the basic building blocks upon which other abstract data types are built. The abstract representation does not necessarily imply a particular implementation. The principal function of the representation is to provide mechanisms that facilitate the composition and hierachical definition of abstract data types.

4) Initialization Section: This section is used to assign an initial value to each variable declared as an abstract data type. The initial value is computed statically. The evaluation of the initialization expression must yield a value that is type-compatible with the variable being initialized. The value must also satisfy any restriction on the type. A violation will result in an exceptional condition that is handled by signaling such a state if a condition handler is defined; otherwise execution is aborted and control is returned to the using program.

5) Operations Section: This section is used to define the semantics of each operation. Each operation consists of a header which contains the operation name and the parameters, and an operation body in which the effects of the operation are described. The behavior of an operation is expressed in terms of initial and final states. The input (output) assertion specifies the initial (final) state by defining the relationship that must hold among the initial (final) state components. The assertion is basically a relational expression and constitutes the fundamental construct of the specification language. Using Hoare's notation, the body of an operation is specified as:

$$\{P\}S\{Q\}$$

where P is the input assertion, Q the output assertion, and S is the list of statements that enforce the input assertion and satisfy the output assertion. Note that the list of statements is not provided in the specification but is rather synthesized by the synthesis system. The operations section provides the semantic definition of the abstract data type. An operation can be either a procedure that operates through side-effects, or a function that returns a value.

The operation header is basically similar to the one described in the interface section except that the type of the operation (function or procedure) is stated explicitly, and the parameters are named. An operation body is composed of three parts: 1) an identifier list part; 2) an optional input assertion part; and 3) an output assertion part. The identifier list part provides the output variables whose values are to be determined to satisfy the output assertion part.

An input assertion part describes the relationship that must hold among input values, and thus must be evaluated upon entry to the operation. The result of the evaluation defines the meaning of the assertion, and may be either true or false. A true value means that the input assertion has been satisfied, and evaluation can proceed to the next assertion; otherwise, the required relation does not hold, and a state violation is posted by notifying the appropriate error handler.

An output assertion part describes the relationship that must hold among the output values in the final state. The evaluation of an output assertion yields a true value if the assertion is satisfied, and a false value otherwise. An output assertion that is not satisfiable implies an error in the semantic specification of the operation.

The exceptional behavior of abstract data types is handled by the restrictions section. A restrictions section consists of a list of zero or more restrictions. Each restriction is made of an operation identifier, and the keyword **signals** followed by a condition handler identifier. Each of the type operations can assume two different states upon interpretation. These two states are distinguished by the truth value of the input assertions. A normal state corresponds to a satisfied input assertion, and an exceptional state corresponds to its nonsatisfaction. Three sequential actions are performed to handle the exceptional state properly. These actions are as follows.

a) detection: the results of the input assertion evaluation provide the means by which the exception is detected;

b) signaling: a predefined signal is generated upon detection; and

c) handling: remedial action is initiated. After completion, control is returned to the end of the operation where the exception occurred.

The specification language supports detection and signaling. The target language implementation must provide exception handling mecahnisms. It is therefore the responsibility of the using programs to handle exceptions that might arise due to the violation of some input assertion.

6) Restrictions Section: This section is used to associate exceptions with operations that can potentially result in an exceptional state. A restriction consists of the operation identifier followed by the keyword **signals** followed by the name of the exception.

7) Tail Section: This section delimits the type specification. The identifier that follows the keyword **end** must match the type identifier that appears in the type header.

B. Declarations

A declaration is the means by which a variable acquires a type. Typing features in a language help in determining the legal context of the use of variables. These variables can be either declared within the using programs or within the type specification. In both cases, the processing of a declaration is performed at compilation time.

1) Variable Declaration in a Using Program: An abstract data type specification acts as a type definition facility within the using programs [16]. The specification must be made available before the type can be used in declarations. A use clause is required in the using programs to provide the necessary information on the generic abstract data type.

The abstract data type declarations involve the following actions:

a) type definition statement: the result of a type definition is to instantiate an abstract data type specification. Each new instantiation is given a different name and involves the association of concrete values with generic parameters;

b) declaration statement: the variable declaration statement associates a specific instantiated type with an identifier;

c) generic type elaboration: the results of evaluating the actual parameters in a declaration statement are bound to the formal parameters. Given that a generic type specification is only a template, the role of the elaboration of a generic type is to emit code that realizes the semantics of the instantiated type specification. Note that code is emitted only once for identical type instantiation;

d) type binding: the elaborated type specification is bound to the variable that appears in the declaration statement; and

e) initialization: the variable being declared is initialized with the value of the expression of the initialization clause in the type specification.

2) Variable Declaration in a Type Specification: An entity in a type specification can be declared as a generic parameter, exported type or operation, record component, representation component, formal parameter of an operation, return value of a function, or local variable. In each case, the declaration provides type and scope information.

Parameters of a type specification are of two types: generic parameters which parameterize the abstract data type; and formal parameters which appear as parameters of the type specification operations. A mode is associated with each formal parameter. There are three modes [17]: **in, out,** and **inout**. An **in** parameter is a parameter that has an initial value that can not be modified within the body of the operation. An **out** parameter is a parameter that must acquire a value within the operation before exit. An **inout** parameter is a parameter that has an initial value and that can be modified. Generic parameters are **in** parameters. The parameters of a function are **in** parameters, and its return value is an **out** parameter. All other parameters acquire an explicit mode.

III. An Illustrative Example

An example of a university library database is used to illustrate the different components of a type specification (see [18] for a complete description). The intent of this example is to focus on the general structure without undue difficulty. A university library database is an object whose behavior is defined by the following operations:

check_out: check out a copy of a book from the library;
return: return a copy of a book to the library;
add_a_book: add a copy of a book to the library;
remove_one_book: remove a copy of a book from the library;
remove_all_books: remove all copies of a book from the library;
list_books: list titles of all books in the library by a particular author;
which_patron: find out which patron (borrower) last checked out a particular copy of a book;
which_books: find out which books are currently checked out by a particular patron.

Fig. 2 exhibits the specification of a library. The line numbers that appear in the figure are provided for reference only; they are not part of the type specification.

A. Discussion

Each of the seven sections of the type specification example is explained in detail below.

1) Type Header: Line 1. The name of the type specified is "library." No generic parameter is associated with this abstract data type.

2) Interface: Lines 2–13. The interface section has three parts. The first part (line 3) lists the types that can be exported. "library" and "list_of_books" are the only exported types in this example. The second part (lines 5–12) lists the operations that can be exported. "add_book," "check_out," "return_book," etc., are the exported operations. The syntax of each operation is described. For example, the "add_book" operation is a procedure and has three parameters: an **inout** parameter of type "library," an **in** parameter of type "book," and an **in** parameter of type "patron." The third part (line 13) is the imports subsection where all user-defined types used within this specification are listed. Two abstract data types, "book" and "patron," are imported. The specification for these two abstract data types appears in the Appendix.

3) Representation: Line 14. The abstract data type "library" is structured as a set with elements of type "book."

4) Initialization: Line 16. A variable declaration always causes initialization of the variable through activation of the initialization clause. Here, a variable of type "library" will be initialized with the empty library value.

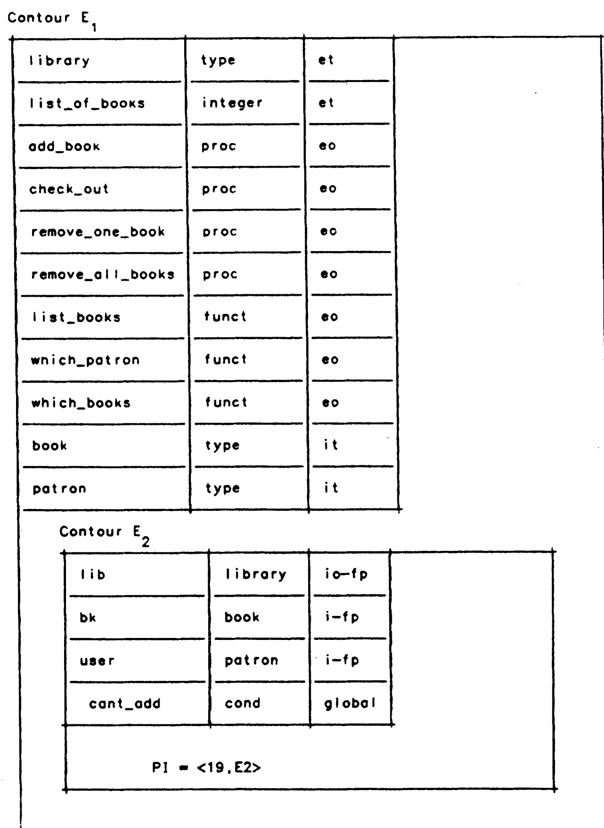
5) Operations: Lines 17–41. This section constitutes the semantic specification of the abstract data type "library." The functionality of the operations is described. Each operation has two components: 1) a header (lines 18, 21, 24, 27, 30, 33, 36, and 39); and 2) a body (lines 19–20, 22–23, 25–26, 28–29, 31–32, 34–35, 37–38, and 40–41). An operation header is made of the operation name, the operation type, and the formal parameters. For example, in the header on line 18, the operation name is "add_book," the operation type is "procedure," and the formal parameters are "lib" which is an **inout** parameter of type "library," "bk" which is an **in** parameter of type "book," and "user" which is an **in** parameter of

```
1    type library;

2    interface
3        types: library, list_of_books;
4        operations:
5            add_book          (inout library; in book; in patron);
6            check_out         (inout library; inout book; in patron, inout patron);
7            return_book       (inout library; inout book; in patron; inout patron);
8            remove_one_book   (inout library; in book; in patron);
9            remove_all_books  (inout library; in book; in patron);
10           list_books        (in library; in patron; in name) returns list_of_books;
11           which_patron      (in library; in book; in patron) returns patron;
12           which_books       (in library; in book; in patron; in patron) returns list_of_books;
13       imports: book, patron;

14   let library       = set (b: book);
15       list_of_books  = set (x: book);

16   initially library = {library: };

17   operations
18   add_book: procedure (inout lib: library; in bk: book; in user: patron)
19        find lib such that lib =  lib' + bk;
20        where patron$is_staff (user) & ~(there exists b in lib: book$equal (b, bk));

21   check_out: procedure (inout lib: library; inout bk: book;  in user: patron; inout pat: patron)
22        find lib such that lib = lib' - bk + book$check (bk, pat) and pat = patron$check (pat)
23        where patron$limit (pat) &  (there exists b in lib: book$equal (b, bk))
             & patron$is_staff (user);

24   return_book: procedure (inout lib:library; inout bk: book; in user: patron; inout pat: patron)
25        find lib such that lib = lib' - bk + book$return (bk) and pat = patron$return (pat)
26        where patron$is_staff (user);

27   remove_one_book: procedure (inout lib: library; in bk: book;  in user: patron)
28        find lib such that lib = lib' -  {library: b in lib': book$equal (b, bk)}
29        where  patron$is_staff (user) ;

30   remove_all_books: procedure (inout lib: library; in bk: book; in user: patron)
31        find lib  such that  lib = lib' - {library: b in lib': book$match (b, bk)}
32        where  patron$is_staff (user);

33   list_books: function (in lib: library; in user: patron; in author_name: name) returns (lb: list_of_books)
34        find lb such that lb = {list_of_books: b in lib': book$author (b, author_name)}
35        where patron$is_staff (user);

36   which_patron: function (in lib: library; in b: book; in user: patron) returns (pat: patron)
37        find pat such that pat = book$borrower (b)
38        where  patron$is_staff (user);

39   which_books: function (in lib: library; in b: book; in user, pat: patron) returns (bl: list_of_books)
40        find bl such that bl = {list_of_books: b in lib': patron$equal (pat, book$borrower (b))}
41        where patron$is_staff (user);

42   restrictions
43        add_book signals cant_add;
44        check_out signals check_out_failure;
45        remove_one_book, remove_all_books, list_books signals no_priviledge;
46        which_books, return_book, which_patron signals no_priviledge;
47   end library.
```

Fig. 2. University library database specification.

type "patron." The operation body consists of an optional list of input assertions, and a list of output assertions. The input assertion lists are: "patron$is__staff & ~(there exists b in lib: book$equal (b, bk))" on line 20; "parton$limit (pat) & (there exists b in lib: book$equal (b, bk))" on line 23; "parton$is__staff(user)" on lines 26, 29, 32, 35, 38, and 41. The output assertion list, "lib = lib' + bk," on line 19 is **lib equals the old value of lib with bk added to it**. Syntactically, an input assertion is introduced by the **where** keyword, and an output assertion is introduce by the **find** keyword.

6) Restrictions: Lines 42–46. The restrictions section is used to take alternative actions in case an input assertion is not satisfied. For example, if the input assertion "patron$is__staff (user)" on line 26 evaluates to false, then the restriction that corresponds to the "return" operation is activated, i.e., the restriction "return **signals** no__privilege" on line 46 is executed.



gp: generic parameter; et: export type; eo: export operation; i-fp: in formal parameter; funct: function; proc: procedure; io-fp: inout formal parameter; global: global within the type; it: import type

Fig. 3. Execution snapshot.

7) Tail: Line 47. This line consists of the keyword **end** and the identifier "library" which matches the identifier that appears after the keyword **type** on line 1.

B. An Execution Snapshot

A simplified version of the contour model [19] is used to display an execution snapshot of the library type specification. The invocation of an abstract data type operation creates a new active environment, which is represented by a contour E_i, a processor state *PI*, and a set of cells. *PI* is an $<ip,ep>$ pair (*ip* = instruction pointer, and *ep* = environment pointer). Each cell represents a variable declaration. Assume the operation "add_book" is invoked. Fig. 3 displays the contour before execution of line 19.

The snapshot shows a total environment made of two nested contours. Contour E_1 corresponds to the global environment of the abstract data type specification shown in Fig. 2. The eleven cells in E_1 correspond to the variables, types, and operations that are known globally in the type specification. Contour E_2 is nested within E_1 and corresponds to the local environment of the operation "add-_book." The local variables are "lib," "bk," "user," and "cant_add," each represented by a cell in E_2. Since "add_book" is the active environment, the value of the processor *PI* points to line 19 of "add_book" and to the contour E_2.

On line 19, there are two references to "lib" and one reference to "bk." All these references are to local variables, and are resolved by searching the name space (i.e., the set of cells) in E_2.

IV. Synthesis System Overview

The system function is to convert an abstract data type specification into a concrete executable program. The specification undergoes incremental refinements until all the specification constructs are completely transformed into programming language constructs. The transition from one intermediate version to another is the result of the application of a set of transformation rules. Intermediate versions constitute partially developed programs composed of contructs of different levels of abstraction. The final version of the program is expressed in a procedural language. The transformation activity is not a monolithic process, but is accomplished in several stages. These stages are as follows.

1) Syntactic and Semantic Analysis: The result of this activity is a parse tree. All overloaded operators are resolved, and the operand attributes are propagated.

2) Type Decomposition: Abstract data types are defined hierarchically. The result of the type decomposition activity is to build a directed acyclic graph that expresses explicitly the hierarchical and structural relationship among the different types of the specification.

3) Specification Transformation: Transformation rules are used to translate constructs of the specification into abstract constructs of an algorithmic intermediate language. There are essentially two activities within this stage. First, the flattening of the parse tree results in a linear form expressed in an intermediate language (intermediate abstract language one or IAL1). Then, a series of refinements and expansions is performed on this linear form to generate an abstract program expressed in another intermediate language (IAL2).

4) Data Structure Selection: A cost analysis is performed, and a set of data structures that implements the abstract data type in the specification is selected. The selection process is designed to choose the implementing data structures associated with a minimum cost that is based on predefined criteria.

5) Data Structure and Algorithm Integration: The selection of a specific set of data structures requires a matching set of concrete algorithms. This integration activity associates the data structures and the algorithms that manipulate them.

6) Program Integration: This stage is the final integration in which the programs implementing the abstract data type are integrated with the main program.

The synthesis process is defined by a function that maps abstract constructs of the specification language into concrete constructs of the target programming language. Let *trf* be this function. The function *trf* is defined by:

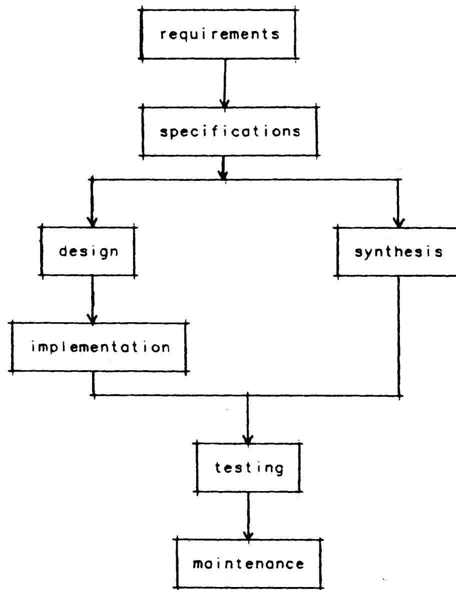


Fig. 4. The synthesis system role in the software life cycle.

1) the input domain which consists of constructs of the specification language. The syntactic aspect of these constructs defines patterns, and the semantic aspect defines predicates on these patterns. Therefore, the input domain of *trf* is a Cartesian product of patterns and predicates. The input domain is a finite domain;

2) the output domain which consists of the constructs of the target language. The output domain is finite also; and

3) the mappings (transformation rules) which associate an abstract construct with an equivalent concrete construct. Given that the input and output domains are finite, the set rules are also finite. This set was constructed manually and made part of the knowledge of the synthesis system.

The application of *trf* requires pattern recognition and predicate satisfaction. Once a pattern is recognized, a set of predicates associated with the pattern is tested. The product of the pattern and the successful predicate will form the input of *trf*. A transformation rule is then selected and applied resulting in another pattern.

A. The Synthesis System and the Software Development Process

The traditional software life cycle consists of six sequential stages: 1) requirements; 2) specification; 3) design; 4) implementation; 5) testing; and 6) maintenance. A software life cycle that integrates the synthesis system would consist of five sequential stages: 1) requirements, 2) specification; 3) synthesis; 4) testing; and 5) maintenance. The synthesis system consolidates and automates the design and implementation stages. These two stages are usually time consuming and introduce specific details that are more related to the programming language than the problem domain. Fig. 4 shows the relationship between the traditional software life cycle and the synthesis system life cycle. The synthesis system is closely related to the Higher Order Software, Inc. functional life cycle [20]. Both approaches attempt to validate the specifications at an early stage of the software development process, and then generate executable code.

B. The Synthesis Process

The semantics of the operations of an abstract data type specification is specified in terms of input and output assertions of the form:

$$\{P(a)\}\ S\ \{R(a, x)\} \tag{1}$$

where

$P(a)$ is the input assertion to be enforced;
$R(a, x)$ is the output assertion to be satisfied; and
S is the list of statements to be synthesized.

Formula (1) is manipulated by the synthesis system in order to derive an implementation S that satisfies P and R. The output assertion $R(a, x)$ has different forms which are:

a) $x = R'(a)$;
b) x <relational operator> $R'(a)$;
c) $R1(x) = R2(a)$;
d) $R1(x)$ <relational operator> $R2(a)$; and
e) $R1(a, x)$ <relational operator> $R2(a, x)$.

The nonterminal <relational operator> denotes $>$, $>=$, $<$, $<=$, or $\hat{=}$. The simplest form is form a). To satisfy the assertion requires finding a value for x such that the equality is satisfied. A trivial way to satisfy the assertion is to set x to R' (a). Form b) requires that the object x have an ordering relation. Ordering is not always available, the fact which implies that the use of form b) will potentially make the use of $R(a, x)$ inconsistent. Even in cases where consistency is safeguarded, the formula $x > R'(x)$ can be satisfied in an infinite number of ways. In programming, specific solutions are of interest.

Form c) is a generalization of form a). In general the assertion cannot be solved unless the number of variables in $R1(x)$ is one, i.e., we have

$$R1(x_1, x_2, \cdots, x_n) = R2(a_1, a_2, \cdots, a_m)$$

where $n = 1$, which implies

$$R1(x_1) = R2(a_1, a_2, \cdots, a_n).$$

This assertion is basically form a), because $R1(x1)$ can be easily reduced to x.

Form d) is a gneralization of forms b) and c), and form e) is the most general form. All the other forms are subcases of form e).

The output assertions of the specification language are of form a). This choice reduces the complexity of the synthesis system, and is consistent with the axiomatic approach as defined in [21]. Therefore, all output assertions in a specification have the following form:

$$x = R(a). \tag{2}$$

Formula (2) can be satisfied using the axiom of assign-

ment rules given below:

$$\{P(a)\}\ x := R(a)\ \{x = R(a)\}. \tag{3}$$

Therefore, the abstract embodiment of the list of statements S is the assignment statement $x := R(a)$. The goal of synthesis system is to decompose this abstract assignment into primitive constructs whose collective effect is equivalent to the abstract construct. That is, the instantiation of (3) is syntactically modified and expanded while its semantics is preserved.

The transformation of (3) depends on the type of the variable name x. Let T be the type of x. T can assume the following values:

1) Boolean:
2) integer;
3) real;
4) string;
5) collection (set or sequence);
6) Cartesian product; and
7) abstract.

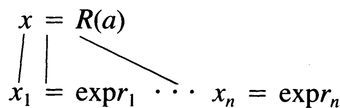
There are several cases that need to be treated separately due to different decomposition schemes. These cases are:

1) The types of x and R (a, x) are primitive unstructured types (i.e., Boolean, integer, real, and string). However, given that $R(a)$ is an expression, it is possible that its operands are of types different from that of its result. There are two subcases:

a) The types of the operands of R (a) are all primitive unstructured types. This kind of assertion can be translated directly into the target language constructs.

b) The types of the operands of R (a) are not (all) of primitive unstructured types. Then, each must be of a type belonging to category 2, 3, or 4. Further decomposition is necessary.

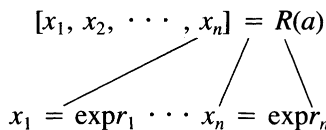
2) x is a collection, and the result of R (a) is a collection. The assertion operator is distributed over all the elements.



3) x is a Cartesian product of several types, i.e.,

$$x = [x_1, x_2, x_3, \cdots, x_n].$$

We thus have



This decomposition is almost identical to the one used for category 2. However, the implementation is different because category 2 involves repetitive constructs, whereas category 3 requires straight line constructs.

4) x is an abstract type defined elsewhere. Its implementation is needed to be able to perform the decomposition. Two tasks are consecutively carried out. First, an interface check is made to determine whether the operation is properly invoked. Next, the parameters, if any, are mapped into their concrete representation to conform to parameter compatibility. This task requires the availability of the interface specification and the implementation of the abstract type in the database.

1) Resolution of Operator Meaning: Overloaded operators have to be assigned a unique meaning. The function of the first stage of the synthesis process is to resolve the meaning of the overloaded operators. Meaning resolution is a simple transformational process. Given an input pattern and a predicate associated with the pattern, an output pattern and a new predicate are generated. The result is syntactically and semantically equivalent to the input pattern. A transformation rule has the form

$$pr\ (x)\ \&\ pa\ (t)\ => pr\ (x')\ \&\ pa\ (t') \tag{4}$$

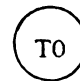
where

pr (x) is the predicate on x;
pa (t) is the pattern of a subtree t;
pr (x') is the new predicate; and
pa (t') is the output pattern.

The purpose of this transformation phase is to resolve operator overloading. Therefore, the transformation rules deal with operators and operands, i.e., expressions of the specification language. Consequently, any complex expression can be decomposed into a finite number of simple constituents represented as trees. These trees define the patterns pa (t) under consideration. The manifestations of the patterns are listed below.

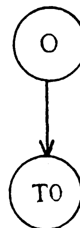
1) Empty Pattern: This pattern corresponds to an empty expression. No transformation is performed on this pattern.

2) Single-Node Pattern:

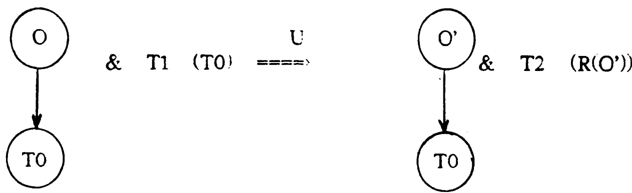


This pattern corresponds to an expression that consists of a single operand and no operator. No transformation is performed on this pattern.

3) Unary Operator Pattern:

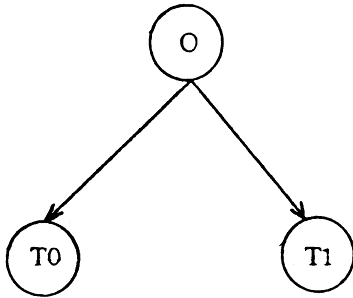


This pattern corresponds to an expression that consists of a unary operator and an operand. The unary operator is represented by the root of the tree, and the operand by the leaf. A transformation U is performed on this pattern. We have:

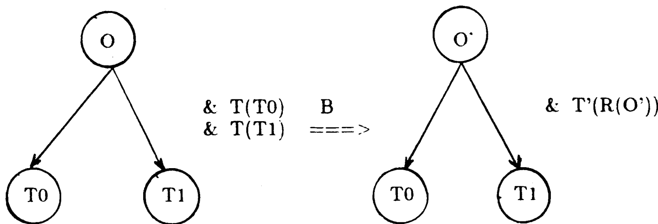


$T1$ and $T2$ are type predicates and R is a function that returns the type of the result of an operator O. O' is the resolved operator. A transformation U is performed on this pattern.

4) Binary Operator Pattern:



This pattern corresponds to an expression consisting of a binary operator and a left and a right operand. A transformation B is performed on this pattern.



5) N-ary Pattern: This pattern is denoted as a tree with a root and N children. An N-ary pattern models all the previous patterns. A transformation N is performed on this pattern.

The predicates used with these transformation rules are type predicates. A type predicate has the form $T(x)$, where x is a variable of some type, and T is a function that returns true if x is of type T, and false otherwise.

The operator meaning resolution transformations are equivalence-preserving functions, because the effect of applying a transformation results only in a renaming of the operator. The renaming function is well-defined, and can be described by listing its domain, its range, and the mapping between the elements of the domain and the elements of the range.

2) Stepwise Transformations: The previous transformational stage reduces the level of abstraction by discriminating among several overloaded operators. However, the amount of detail provided and the abstractness of the constructs are left unchanged. It is the function of the second stage of the synthesis to transform the abstract constructs into more concrete constructs. Generally, a transformation of an abstract construct results in a refinement, thus introducing more constructs. These new constructs may be themselves subject to further transformation. The process is applied recursively until all the constructs are concrete.

V. Transformation Rules

A transformation is selected and applied whenever a pattern is recognized and a predicate is satisfied. Therefore, the description of these rules will consist of a pattern, a predicate, and a transformation template. The template constitutes the body of a parameterized procedure. The parameters of the procedures correspond to the operands of the operator being transformed. The instantiation of the template generates a set of new patterns and new predicates. New patterns that are abstract are added to a list of candidates for further transformation. There are four types of patterns in a transformation template. These patterns are as follows.

1) zero-ary patterns: these patterns consist of a keyword of IAL2. Zero-ary patterns are used for grouping other patterns.

2) unary patterns: these patterns consist of a keyword of IAL2 and an operand.

3) binary patterns: these patterns consist of a keyword of IAL2 and two operands.

4) ternary patterns: these patterns consist of a keyword of IAL2 and three operands.

The operands appearing in transformation template pattern are either operands that appear in the pattern being transformed, in which case they bear the same identifier, or they are temporary operands newly introduced. Identifiers of temporary operands are of the form "*t__i*". Some of the patterns of the transformation templates are prefixed with the symbol "(*)". This symbol indicates that the pattern requires more refinement.

The transformation rules are classified into the following four categories.

1) sequence transformation rules: these rules are applied to operators that manipulate sequence and elements of sequences.

2) set transformation rules: these rules are applied to operators that manipulate sets and elements of sets.

3) plex transformation rules: these rules are applied to operators that manipulate plexes and components of plexes.

4) miscellaneous transformation rules: there is only one rule in this category.

A type that is structured as a sequence inherits all the sequence operations. A type that is structured as a set inherits all the set operations. A type that is structured as a plex inherits all the plex operations.

The operation "add__book" of the example shown in Fig. 3 is used in this section to illustrate the different phases of the synthesis process. The operation is transformed through five stages in the next subsections.

A. Stage One

The operation is transformed into overloaded constructs of IAL1. The transformed code is shown in Fig. 5. Line numbers were added for reference. Patterns that have a

```
1    begin_operation          add_book
2    begin_input_assertion
3        function_call        patron$is_staff   1      t__01
4        actual_parm          user
5        constructor          b                 lib
6        function_call        book$equal        2      t__02
7        actual_parm          b
8        actual_parm          bk
9        there_exists         t__02
10       not                  t__03             t__02
11       and                  t__04             t__01  t__03
12   end_input_assertion      t__04
13   begin_output_assertion
14*      add                  t__05             lib    bk
15*      assert               t__06             lib    t__05
16   end_output_assertion
17   end_operation            add_book
```

Fig. 5. The "add_book" operation representation in generic IAL1.

```
1    begin_operation          add_book
2    begin_input_assertion
3        function_call        patron$is_staff   1      t__01
4        actual_parm          user
5*       constructor          b                 lib
6        function_call        book$equal        2      t__02
7        actual_parm          b
8        actual_parm          bk
9*       there_exists         t__02
10       not                  t__03             t__02
11       and                  t__04             t__01  t__03
12   end_input_assertion      t__04
13   begin_output_assertion
14*      set_element_radd     t__05             lib    bk
15*      set_assert           t__06             lib    t__05
16   end_output_assertion
17   end_operation            add_book
```

Fig. 6. The "add_book" operation representation in specific IAL1.

star (*) after the line number are candidates for the first transformational stage. For example, on line 14, the operator "add" is an overloaded operator. The abstract nature of the operator subjects the pattern "add *t*___05 lib bk" to further transformation. The predicate "*S*(lib) = set" is satisfied in this case, and a new pattern ("set__element__radd *t*___05 lib bk" on line 14 of Fig. 6) is generated.

B. Stage Two

The constructs that need transformation are identified and transformed into specific IAL1 code. Fig. 6 shows the result of transforming constructs of Fig. 5.

C. Stage Three

The constructs of IAL1 that need transformations are identified (starred lines in Fig. 6). These constructs are transformed into IAL2 constructs. For example, on line 14 of Fig. 6, the operator "set__element__radd" is an abstract operator that requires refinement. The abstract nature of the operator subjects the pattern "set__element__radd *t*___05 lib bk" to further transformation. The predicate "*S*(*t*___05) = *S*(lib) = set $ *T*(*t*___05) = *T*(lib) & *T*(bk) = *T*(*C*(lib))" is satisfied in this case. The transformation template corresponding to this pattern and this predicate is instantiated. The result of these transformations is shown in Fig. 7.

D. Stage Four

Data structure selection is performed at this stage. This process uses the results of a previous analysis of the behavior of the abstract data type operations to select concrete data structures. A knowledge base [22], [23] about the complexity of different algorithms on several implementations of each abstract data structure is used by the synthesis system to select algorithms and data structures that minimize the cost complexity [24]. The information contained in the knowledge base was obtained through analysis of algorithms and data structures that were implemented using the random access machine model of [25].

Other optimization techniques for high-level data structures and constructs [26], [27] are being investigated for possible implementation.

```
1          begin_operation              add_book
2          begin_input_assertion
3              function_call            patron$is_staff  1      t__01
4              actual_parm              user
5*             enumerate_set            b                lib
6                function_call          book$equal       2      t__02
7                actual_parm            b
8                actual_parm            bk
9                if_test                t__02
10               exit                   l__01
11             end_enumerate_set
12             not                      t__03            t__02
13 l__01:      and                      t__04            t__01  t__03
14         end_input_assertion          t__04
15         begin_output_assertion
16             begin_set_element_radd
17*              copy_set               t__05            lib
18*              radd_set_element       t__05            t__04  bk
19             end_set_element_radd
20*            set_assert               t__06            lib    t__05
21         end_output_assertion
22         end_operation                add_book
```

Fig. 7. The "add_book" operation representation in IAL2.

```
add_book: entry (lib,bk,user);
     dcl   cant_add                      condition;
     dcl   tp__01                        fixed bin;

     t__01 = patron$is_staff (user);
     call rep$set_size (tp__01,lib);
     t__02 = false;
     do while (tp__02 > 0);
          call rep$enumerate_set (b,lib);
          t__02 = book$equal (b,bk);
          if   t__02
          then go to l__01;
          else;
          tp__01 = tp__01 - 1;
          end;
     l__01:
     t__03 =  t__02;
     t__04 = t__01 & t__03;
     if   t__04
     then do;
          begin;
            call rep$copy_set (t__05,lib);
            call rep$radd_set_element (t__05,bk);
          end;
          lib = t__05;
          end;
     else signal cant_add;
     return /* add_book */;
```

Fig. 8. The "add_book" operation implementation in PL/I.

E. Stage Five

This is the last stage in the transformation process. Constructs of IAL2 are translated directly into PL/I constructs. Fig. 8 shows the result of this stage.

VI. Summary

The transformational approach to programming is intended to free the software engineer from details that are irrelevant to the problem domain. The abstract data type specification language described in this paper can aid in minimizing the effort in the software life cycle. The synthesis system described in this paper uses the programming paradigm of stepwise refinement to automatically transform an abstract specification into an executable program. The synthesis process is not a monolithic process. Several automated stages and successive refinements are required to completely generate a concrete program. A set of transformation rules is associated with each stage of the synthesis process. These transformation rules were described, and an example to illustrate their selection and application was given. Experience with the system includes specification and synthesis of several moderately sized abstract data types, such as a symbol table for a block-structured language, a university library database, and a resource manager [28]. Qualitative and quantitative analyses of the specifications and their implementing programs reveal a definite advantage in using the synthesis approach to software development. The implementing programs are not as efficient at this moment as hand-coded programs; however, reliability, and early availability for testing have a greater impact on the software life cycle than execution efficient.

The existing synthesis system generates PL/I and Pascal code from abstract specifications. Future work includes the enhancement of the synthesis system to support code emission for additional procedural languages. Currently, research is being performed involving knowledge-based optimization of the generated code, code generation in Ada® and C, automatic test generation from specifications, and the rehosting of the synthesis system on a Vax system.

®Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

Appendix

Patron Specification

```
 1  type patron;

 2  interface
 3    types: patron;
 4    operations:
 5       equal          (in patron; in patron) returns (boolean);
 6       is_staff       (in patron) returns (boolean);
 7       limit          (in patron) returns (boolean);
 8       empty           returns (patron);
 9       check          (in patron) returns (patron);
10       return         (in patron) returns (patron);
11       create_patron  (in name; in integer; in boolean) returns (patron);
12    imports: name;

13  let patron = tuple (patron_name: name,
14                      book_count:   integer,
15                      book_limit:   integer,
16                      staff:        boolean);

17  initially patron = [patron: name$empty, 0, 0, false];

18  operations

19  equal: function (in p1, p2: patron) returns (e: boolean)
20         find e such that e = name$equal (p1.patron_name,p2.patron_name);

21  is_staff: function (in pat: patron) returns (e: boolean)
22         find e such that e = pat.staff;

23  limit: function (in pat: patron) returns (e: boolean)
24         find e such that e = pat.book_limit >= pat.book_count;

25  empty: function returns (pat: patron)
26          find pat such that pat = [patron: name$empty, 0, 0, false];

27  check: function (in p1: patron) returns (p2: patron)
28          find p2 such that p2 = [patron: p1.patron_name, p1.book_count + 1, p1.book_limit, p1.staff];

29  return: function (in p1: patron) returns (p2: patron)
30          find p2 such that p2 = [patron: p1.patron_name, p1.book_count - 1, p1.book_limit, p1.staff];

31 create_patron: function (in n: name; in i: integer; in s: boolean) returns (p: patron)
32          find p such that p = [patron: n, 0, i, s];

33  end patron.
```

Book Specification

```
 1  type book;

 2  interface
 3    types: book;
 4    operations:
 5       equal      (in book; in book) returns (boolean);
 6       match      (in book; in book) returns (boolean);
 7       author     (in book; in name) returns (boolean);
 8       make_book  (in name; in string; in string; in integer) returns (book);
 9       borrower   (in book) returns (patron);
10       check      (in book; in patron) returns (book);
11       return     (in book) returns (book);
12    imports: patron, name;

13  let book = tuple (author_name: name,
14                    title:       string,
15                    call_number: string,
16                    copy_number: integer,
17                    checked:     boolean,
18                    user:        patron);

19  initially book = [book: name$empty, "*", "*", 0, false, patron$empty];

20  operations

21  equal: function (in b1, b2: book) returns (e: boolean)
22         find e such that e =  (match (b1, b2)) &  (b1.copy_number = b2.copy_number)

23  match: function (in b1, b2: book) returns (e: boolean)
24         find e such that e = (name$equal (b1.author_name,b2.author_name) &
25                               (b1.title = b2.title) &
26                               (b1.call_number = b2.call_number);

27  author: function (in b: book; in n: name) returns (e: boolean)
28          find e such that e = name$equal (b.author_name, n);

29  make_book: function (in n: name; in c1, c2: string; in i: integer) returns (b: book)
30          find b such that b = [book: n, c1, c2, i, false, patron$empty];

31  borrower: function (in b: book) returns (pat: patron)
32          find pat such that pat = b.user;

33  check: function (in b1: book; in pat: patron) returns (b2: book)
34          find b2 such that b2 = [book: b1.author_name, b1.title, b1.call_number, b1.copy_number,
                                    true, pat];

35  return: function (in b1: book) returns (b2: book)
36          find b2 such that b2 = [book: b1.author_name, b1.title, b1.call_number,
37                                        b1.copy_number, false, patron$empty]

38  end book.
```

Acknowledgment

We would like to thank G. Riccardi for his contributions to the initial phase of this project, and T. Walker for his support.

References

[1] B. K. Boehm, 'Software engineering: R&D trends and defense needs,'' in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: M.I.T. Press, 1979, pp. 44-86.

[2] B. Belkhouche, ''Automatic synthesis of abstract data type implementations from abstract specification,'' Ph.D. dissertation, Dep. Comput. Sci., Univ. Southwestern Louisiana, May 1983.

[3] B. Belkhouche and J. E. Urban, ''An executable specification language for abstract data types,'' *RAIRO Technique et Science Informatiques*, vol. 3, no. 4, pp. 247-251, June-Aug. 1984.

[4] B. Belkhouche, ''Executable specifications of abstract data types: The university library data base example,'' presented at the Int. Workshop Software Specification and Design, Orlando, FL, Mar. 30, 1984.

[5] M. Shaw, *ALPHARD: Form and Content.* New York: Springer-Verlag, 1981.

[6] W. A. Wulf, R. L. London, and M. Shaw, ''An introduction to the construction and verification of ALPHARD programs,'' *IEEE Trans. Software Eng.*, vol. SE-2, pp. 253-265, Dec. 1976.

[7] V. Berzins, ''Abstract model specification for data abstraction,'' Ph.D. dissertation, Massachusetts Inst. Technol. Cambridge, Rep. MIT LCS TR-221, July 1979.

[8] C. A. R. Hoare, ''Notes on data structuring,'' in *Structured Programming*, O.-J. Dahl, E. W. Dijsktra, and C. A. R. Hoare, Eds. New York: Academic, 1972, pp. 271-281.

[9] J. A. Goguen and J. J. Tardo, ''An introduction to OBJ: A language for writing and testing formal algebraic specifications,'' in *Proc. Conf. Specifications of Reliable Software*, IEEE Comput. Soc., Apr. 1979, pp. 75-84.

[10] J. V. Guttag, ''The algebraic specification of abstract data types,'' *Acta Inform.*, vol. 10, no. 1, pp. 27-52, 1978.

[11] B. Liskov and S. Zilles, ''Specification techniques for data abstractions,'' in *Proc. Int. Conf. Reliable Software, SIGPLAN Notices*, vol. 10, no. 6, pp. 72-87, June 1975.

[12] L. Flon and J. Misra, ''A unified approach to the specification and verification of abstract data types,'' in *Proc. Conf. Specifications of Reliable Software*, IEEE Comput. Soc., Apr. 1979, pp. 162-169.

[13] D. Bert, ''La programmation generique,'' Ph.D. dissertation, Univ. Scientifique et Medicale de Grenoble, France, June 1979.

[14] M. E. Majster, ''Treatment of partial operations in the algebraic specification technique,'' in *Proc. Conf. Specifications for Reliable Software*, IEEE Comput. Soc., Apr. 1979, pp. 190-197.

[15] C. A. R. Hoare, ''Proof of correctness of data representations,'' *Acta Inform.*, vol. 1, 1972, pp. 271-281.

[16] D. Jensen and N. Wirth, *Pascal User Manual and Report*, 2nd ed. New York: Springer-Verlag, 1977.
[17] J. D. Ichbiah, J. P. Barnes, J. C. Heliard, B. Krieg-Brueckner, O. Roubine, and B. A. Wichmann, "Part B: Rationale for the design of the Ada programming language," *SIGPLAN Notices*, vol. 14, no. 6, June 1979.
[18] R. A. Kemmerer, "Testing formal specifications to detect design errors," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 32-43, Jan. 1985.
[19] J. B. Johnston, "The contour model of block structured processes," General Electric Tech. Inform. Series 70-C-366, Oct. 1970.
[20] M. Hamilton and S. Zeldin, "The functional life cycle model and its automation: USE.IT," *J. Syst. Software*, vol. 3, no. 1, pp. 25-62, Mar. 1983.
[21] C. A. R. Hoare, "An axiomatic approach to computer programming," *Commun.* ACM, vol. 12, no. 10, pp. 576-580, 583, Oct. 1969.
[22] D. Barstow, "Automatic construction of algorithms and data structures using knowledge base of programming rules," Ph.D. dissertation, Stanford Univ. Stanford, CA, Rep. AIM-308, Nov. 1977.
[23] E. Kant, "Efficiency considerations in program synthesis: A knowledge-based approach," Ph.D. dissertation, Stanford Univ., Stanford, CA, Rep. AIM-331, Sept. 1979.
[24] J. R. Low, "Automatic data structure selection: An example and overview," *Commun.* ACM, vol. 21, no. 5, pp. 376-385, May 1978.
[25] A. V. Aho, J. E. Hopcroft, and J. D. Ulmann, *Design and Analysis of Algorithms*. Reading, MA: Addison-Wesley, 1974.
[26] R. B. K. Dewar, A. Gand, S.-C. Liu, J. T. Schwartz, and E. Schonberg, "Program refinement as exemplified by the SETL representation sublanguage," *ACM Trans. Program. Lang. Syst.*, pp. 27-49, July 1979.
[27] J. Earley, "High level iterators and a method for automatically designing data structure representation," *J. Comput. Lang.*, vol. 1, pp. 321-342, 1976.
[28] B. Belkhouche, "Several examples of specifications of abstract data types," Dep. Comput. Sci., Tulane Univ., New Orleans, LA, Tech. Rep., May 1984.

Boumediene Belkhouche (S'80-M'82) received the Ph.D. degree in computer science from the University of Southwestern Louisiana, Lafayette, in 1983.

He is now an Assistant Professor of Computer Science at Tulane University, New Orleans, LA. His areas of interest include specification and implementation of abstract data types, specification languages, design and implementation of programming languages, software engineering, and software prototyping. In the past three years, he has been involved in the design and implementation of rapid prototyping environment based on abstract data types.

Dr. Belkhouche is a member of the Association for Computing Machinery. He is a member of the IEEE Computer Society Distinguished Visitors' Program for 1985-1986.

Joseph E. Urban (M'82) received the B.S. degree from Florida Institute of Technology, Melbourne, in 1973, the M.S. degree from the University of Iowa, Iowa City, in 1975, and the Ph.D. degree from the University of Southwestern Louisiana, Lafayette, in 1977, all in computer science.

He is now an Associate Professor of Computer Science, responsible for the software engineering program, at the University of Southwestern Louisiana. His research interests are software engineering, specification languages, test data generation, and computer languages.

Dr. Urban is a member of the Association for Computing Machinery. His Ph.D. dissertation was selected by the ACM Doctoral Forum as one of the four best computer science theses produced in 1977-19778. He is on the Editorial Board of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING and a member of the IEEE Computer Society Governing Board.