Abstract design and program translator: New tools for software design

by J. L. Archibald
B. M. Leavenworth
L. R. Power

Abstract Design And Program Translator (ADAPT) is an integrated set of tools and approaches for the design and development of software systems. Together they include a module specification language and a system design language for specifying module interfaces and interconnections. This paper explains some of their major features and illustrates their use in the design of some examples—a set of reusable software components and a generalized editor system. Benefits of the ADAPT approach are discussed, emphasizing executable design and modifiability.

Discussed in this paper is a new approach to software design that consists of two primary tools: (1) a system definition language, called the External Structure, which supports programming in the large, and (2) a specification language, the ADAPT programming language, for specifying the semantics of individual modules. ADAPT is our acronym for Abstract Design And Program Translator.

We discuss first general concepts of ADAPT and its components. These concepts are then illustrated using examples of module and system design. We compare and contrast ADAPT methodology and current practice and show ways of improving the specification process in general and module and system specification in particular. We conclude with a discussion of experience, design tradeoffs, analysis tools, and compatibility of ADAPT with existing code.

General concepts and components

ADAPT is based on the use of two languages. The External Structure language provides for descriptions of all modules contained within a system, giving their allowable interfaces and the interconnections between modules. The External Structure allows the separate compilation of individual modules and becomes a repository of design information at different stages in the design process.

The ADAPT specification language provides for semantic specification of individual modules. It is similar to such other data abstraction languages as CLU,[1] Ada,[2] MESA,[3] and Euclid.[4] The specification language has facilities for defining three kinds of modules, corresponding to procedural, data, and iterative (control) abstractions.

There are two fundamentally different approaches to the specification of systems. In the first, the specification—whether in English or some formal or semiformal representation—and the implementation are separate. The problem in this case is

showing the equivalence of the two representations. This may be accomplished by proving program correctness or by testing. Further, it is difficult to keep these two descriptions synchronized during the software life cycle. In the other approach, there is only one specification, which is then transformed to an executable program by a translator. This ensures semantic equivalence. ADAPT follows the latter approach, even though it has a separate language for high-level system descriptions.

In the ADAPT approach, the tools include a translator that converts ADAPT programs to a target language, currently PL/I. The translator reads in a system specification written in the External Structure language and a module specification written in the ADAPT specification language. The module specification is processed with respect to the system specification. Processing consists of strong type checking, in addition to the translation to the target language. A variety of outputs may be obtained to assist in the design process.

The ADAPT language

ADAPT is an algorithmic language for defining the detailed semantics of *modules*. In addition to control structures comparable to those of other high-level languages, ADAPT provides facilities for handling abstract data. The three kinds of modules which can be defined are the following:

- PROCEDURE is like procedures in other languages, but parameters may be user-defined data types, in addition to the data types provided by the language.
- CAPSULE is used to specify a data abstraction and consists of an internal data representation and operations (called *encapsulated procedures* or *iterators*) that manipulate the internal representation.
- ITERATOR is used to specify how elements of an abstract data collection are obtained so that actions on the elements can be programmed independently.

An ADAPT procedure or iterator can declare variables of built-in or user-defined data types, can invoke operators of built-in or user-defined data types, and can call procedures and invoke iterators. There are no nested procedures, but a capsule can have nonexported (local) procedures and iterators, in addition to the exported operators of the data type. There are no global variables.

ADAPT supports the separate compilation of modules. References to externally defined names of data type, procedures, and iterators are resolved by the External Structure.

The ADAPT specification language provides for semantic specification of individual modules.

To describe the features of ADAPT, we note its differences from such current languages as PL/I or Pascal as we present the elements of the ADAPT language.

Primitive types. We begin with the following primitive data types and type constructors of ADAPT:

```
INT
BOOL
STRING
CHAR
NULL
```

INT is similar to the type *integer* of Pascal and to FIXED BIN (31) of PL/I. BOOL is similar to type *boolean* of Pascal, where the keywords TRUE and FALSE in ADAPT correspond to Pascal's *true* and *false*. PL/I uses bit strings to represent truth values. STRING in ADAPT differs from Pascal and PL/I in that in ADAPT it has unbounded length. CHAR is for fixed-length strings and corresponds to PL/I CHARACTER. NULL is a type having the single literal value NIL. NIL should not be confused with the built-in function NULL in PL/I or the value **nil** in Pascal, both of which represent a pointer pointing to nothing. There are no explicit pointers in ADAPT. All of the objects in these data types are *immutable*, i.e., objects with values that never change.

Type constructors. A type constructor is a parameterized type definition. This is also referred to as a *generic type*. A particular data type is obtained when specific values are supplied for the parameters as follows:

```
RECORD  {s1:t1,...,sn:tn}
ONEOF   {s1:t1,...,sn:tn}
ARRAY   {type-spec,size}
```

The RECORD type is similar to a record in Pascal or structure in PL/I. That is, one or more field names

Use of compound names resolves conflicts among identical operation names in different data types.

are specified, together with the data types associated with each field, as in the following example:

```
DCL R RECORD{VALUE:BOOL,COUNT:INT};  /* Record declare */
... R.VALUE ...          /* Access of Record component */
```

The ONEOF type is a discriminated union similar to the variant record of Pascal. In ADAPT, any type may be associated with a field name, whereas in Pascal the variants are all record types. PL/I does not have a counterpart to the ONEOF. The following is an example of ONEOF declaration:

```
DCL U ONEOF{EMPTY:NULL,NONEMPTY:STRING}
```

Access to the value of a ONEOF is through the control structure SELECT TAG.

The ARRAY type in ADAPT has a fixed length similar to Pascal. Dynamic arrays can be defined as user-defined types, using the fixed-length ARRAY as a building block. The syntax of subscripted variables is the same as PL/I. The following is an example of the ARRAY type in ADAPT:

```
DCL A ARRAY{INT,100};                /* Array declare */
... A(24) ...            /* Access of array component */
```

Initialization of data types. Instances of primitive data types are given their values by the appearance of literals of the proper form. Type constructors must be initialized with a CREATE expression of the following form:

```
type-spec•CREATE ( initialization-list )
```

Here, type-spec may be a RECORD, ONEOF, or ARRAY type. The notation type-spec•CREATE (requesting invocation of the CREATE operator for the type-spec data type) is an example of a compound name. This consists of a type designation, a dot (•) and the operation (in this case CREATE). Use of compound names resolves conflicts among identical operation names in different data types.

For the RECORD type, the initialization list may be empty or consist of one or more name:value pairs as in the following example:

```
X = RECORD{VALUE:BOOL,COUNT:INT}•CREATE(VALUE:TRUE,COUNT:0);
```

Any fields not initialized must subsequently be assigned values. The ONEOF type must be initialized with a proper tag:value pair, as shown in the following example:

```
Y = ONEOF{EMPTY:NULL,NONEMPTY:STRING}•CREATE(EMPTY:NIL);
```

The initialization list for the ARRAY type must be empty, as in the following example:

```
Z = ARRAY{INT,100}•CREATE();
```

RECORDs and ARRAYs are instances of mutable data, where individual components of the construction may be updated by program execution. The tag field of the ONEOF constructor is immutable. Newly created instances of mutable data are accessed through an indirect reference.

Expressions. As in other languages, ADAPT expressions are either operands or applications of allowable operators to operands. Operands may be literals or variables (which may be simple identifiers or selections from instances of constructor types). When an operand object is created, it persists as long as it is referenced in the program. Explicit deallocation is not allowed. This is an object-oriented model of data. There are no *dangling references* in ADAPT, as there are in PL/I or Pascal. Operators are either external- or encapsulated-function invocations, or they are either arithmetic or relational infix operators. Encapsulated functions are referred to by compound names, as in the following examples:

```
Y = QUEUE•PUT(MYQUEUE,QUEUE•GET(HISQUEUE));
Z = X+STACK{INT}•TOP(MYSTK);
```

Statements. ADAPT has conditional statements (IF...THEN... and IF...THEN...ELSE...), loop statements (DO WHILE... and DO UNTIL...), a compound statement (DO...END), and a SELECT statement. These

behave as they do in PL/I. A LEAVE statement differs from PL/I. In ADAPT a LEAVE statement terminates execution of the enclosing loop (DO WHILE... or DO UNTIL...), whereas in PL/I this may also be used to terminate execution of an enclosing compound (DO...END). ADAPT provides an assignment statement, RETURN statement, and constructs for function and procedure invocations. There is no GO TO statement.

An EQUATE statement in ADAPT establishes a correspondence between an identifier and either a literal or a type specification. (The **const** and **type** declarations of Pascal are similar to an EQUATE statement.) In PL/I, similar effects require the use of the PL/I Preprocessor. EQUATE is used in the following example:

```
EQUATE SLOGAN "NOW IS THE TIME";
EQUATE REC RECORD(NAME:STRING,FLAG:BOOL);
```

When an operand object is created, it persists as long as it is referenced in the program.

Finally, ADAPT has a declaration statement that has the same syntax as PL/I but can also be used to declare variables of user-defined data types, as shown in the following examples:

```
DECLARE STR STRING;    /* Declares STR to be an unbounded STRING */

DCL ST CHAR(10);       /* Declares ST to be a fixed length       */
                       /* string of length 10                    */

DCL STK MY_STACK(INT) /* Declares STK to be a user defined       */
                       /* data type construction                 */
```

Procedures. ADAPT procedures have a syntax similar to PL/I, with the difference that a procedure header in ADAPT has the following form:

```
proc-name : PROC (s1:t1,...,sn:tn);
```

Here, the types of the parameters are specified in-line and are thus similar to Pascal. A major semantic difference between ADAPT procedures and those of conventional languages is that ADAPT parameters can be of a type that is user-defined, in addition to the primitive data types of the language.

Capsules. Capsules are implementations of user-defined data types, which we illustrate by the following example for complex integers:

```
COMPLEX: CAPSULE EXPORTS (CREATE,ADD);

  EQUATE REP RECORD(RE:INT,IM:INT);

  CREATE: PROC (X:INT,Y:INT) RETURNS (*);
    RETURN(REP•CREATE(RE:X,IM:Y));
  END CREATE;

  ADD: PROC (X:*,Y:*) RETURNS (*);
    RETURN(REP•CREATE(RE:X.RE+Y.RE,IM:X.IM+Y.IM));
  END ADD;

END COMPLEX;
```

The first line is the header, which defines COMPLEX as a capsule, and defines CREATE and ADD as names of operations (encapsulated procedures) to be exported (i.e., made visible outside the capsule). The second line (i.e., the EQUATE REP statement) defines the internal data representation of the capsule to be a record with two fields, RE and IM, both of type INT.

The CREATE procedure has two parameters of type INT, which are the components of the complex number to be created. The asterisk (*) in the header indicates that a complex number is to be returned as the result of invoking the CREATE procedure, but that the object produced inside the procedure is a record (i.e., the REP object). Therefore, the argument of the RETURN statement is a record created with its fields initialized by the values of the arguments X and Y. The asterisk in a RETURNS clause represents a change of interpretation from concrete representation (REP) to abstract representation (COMPLEX).

The ADD procedure has two parameters with the asterisk notation. This indicates that they are considered to be complex numbers outside of the procedure and records inside the procedure. The RETURN statement, therefore, creates a record and initializes its fields with the sums of the components of the arguments to the procedure. Here in a parameter position, the asterisk represents a change of interpretation from the abstract to the concrete.

Iterators. An iterator is a module that implements a form of control abstraction. It allows the designer to hide the details of sequencing through an aggregate data abstraction by concealing the representation of

the abstraction and the sequencing algorithm. Like procedures, iterators may be encapsulated within a data abstraction, or they may exist as an abstraction in their own right. An iterator produces the elements of the aggregate object one at a time. It therefore must retain state information to produce the next element when required. The following example illustrates an iterator module:

```
ONE_TO_N: ITERATOR (N:INT) YIELDS (INT);
  DCL I INT;
  I = 1;
  DO WHILE (I<=N);
    YIELD(I);
    I = I + 1;
  END;
END ONE_TO_N;
```

This iterator produces the integers from 1 to N, where N is the argument supplied to the iterator when it is invoked. An iterator is invoked using a FOR statement, as follows:

```
DCL A ARRAY{INT,10};
FOR J:INT IN ONE_TO_N(10);
  A(J) = 0;
END;
```

This initializes all the elements of A to 0.

The External Structure language

Systems are described by use of the External Structure language. In this language, a system is a collection of modules and their allowable interconnections. The definition of a module includes the name of each interface (entry point) to the module, as well as the types of the parameters and return value for the interface (a functional type).

A module in an External Structure is viewed as an abstract entity. It is only by means of ADAPT specifications, as discussed in the previous section, that concrete details about a module are given. Modules may be procedures, iterators, or data types. For procedures and iterators, the module interface definition is the functional type of the module. For these modules, there is only one interface. For example, the integer MAX function is described as follows:

```
MAX(INT,INT) -> INT
```

Procedures are not required to have a return value, and their functional type specification has abbreviated syntax. A WRITE procedure to transmit a STRING to a printer has the following syntax:

```
WRITE(STRING)
```

ITRAVERSE, an iterator for a list of integers, is defined as follows:

```
ITRAVERSE ITERATOR(LIST{INT}) => INT
```

When the module is a data type, it is described by a set of operators, that is, by encapsulated procedures and iterators. This set is called the DEFINES list for the module. A data type MESSAGE_QUEUE, where MESSAGE is another data type, has the following specification:

Systems are described by use of the External Structure language.

```
TYPE MESSAGE_QUEUE
  DEFINES
    ( CREATE   () -> MESSAGE_QUEUE
      GET      (MESSAGE_QUEUE) -> MESSAGE
      PUT      (MESSAGE_QUEUE,MESSAGE)
      IS_EMPTY (MESSAGE_QUEUE) -> BOOL )
```

In addition to the interface definitions, the dependencies of the module are listed and refer to the modules which are used (callable) by the module being described. This list is called a USING list. As an example, if module TAB requires a LIST of GABs, the INPUT_STACK facility, and the MESSAGE_QUEUE, this is written as follows:

```
TYPE TAB
  DEFINES
    ( ... )
  USING
    ( LIST{GAB}
      INPUT_STACK
      MESSAGE_QUEUE )
```

Although USING lists are required for the compilation of a module, they can be deferred or abbreviated at the early stages of design. This allows the designer to defer representation decisions and allows the module specification to guide the system specification.

The External Structure for a system consists of a set of descriptions of this form for all modules in the system. This is used when compiling an ADAPT source module, which is said to be compiled with respect to a particular External Structure.

Data types may be generic. In this case, they take other types as parameters, as illustrated by the following notation:

```
name1 {name2, ... }
```

This notation represents the instantiation of a generic data type. The notation further represents a binding of the generic data type with its constituent data types, as illustrated by the following examples:

```
SET{INT}            (set of integers),
SET{TABLE}          (set of tables),
SET{TREE{STRING}}   (set of trees of strings).
```

When processing a module specification, the translator reads the External Structure and the ADAPT source module. The translator verifies that all source module references are consistent with External Structure specifications and other declarations within the module. At the same time, the code generator produces appropriate target language source code. Code generation also exploits the External Structure because the target language descriptions of the run-time environment are adapted from the appropriate descriptions in the External Structure. Figure 1 shows the components and interrelationships of ADAPT.

When processing a module specification, the translator reads the External Structure and the ADAPT source module.

External Structure *graphs* are an additional output that reduces each module to a simple "black box" and displays the interdependences of modules in a graphic form. An example of an External Structure graph for a small system is given in Figure 2.

Examples

Module design. We now present the specifications for SET and SEQ, two abstractions derived from the built-in constructors of PDL.[6] PDL was originally a procedural pseudo-code language to which data abstractions were later added. SET and SEQ, as well as STACK, LIST, and QUEUE, are built-in abstractions of PDL.

First, we give ADAPT External Structure specifications for SET and SEQ:

```
TYPE SET{T:TYPE}
  WHERE  ( T HAS (EQ(T,T) -> BOOL) )
  DEFINES
    ( CREATE   ()          -> SET{T}
      INSERT   (SET{T},T)
      DELETE   (SET{T},T)
      EACH ITER(SET{T})    => T
      EMPTY    (SET{T})    -> BOOL )
  USING
    ( FLEX BOOL )

TYPE SEQ{T:TYPE}
  DEFINES
    ( CREATE   ()          -> SEQ{T}
      RESET    (SEQ{T})
      READ     (SEQ{T})    -> T
      WRITE    (SEQ{T},T)
      EMPTY    (SEQ{T})    -> BOOL )
  USING
    ( FLEX BOOL )
```

We now present the ADAPT module specification source for SET:

```
SET:
  CAPSULE{T:TYPE}
    EXPORTS (CREATE,INSERT,DELETE,EACH,EMPTY)
    WHERE T HAS (EQ PROC (T,T) RETURNS (BOOL));
    EQUATE REP RECORD{SIZE:INT,ELT:FLEX{T}};
```

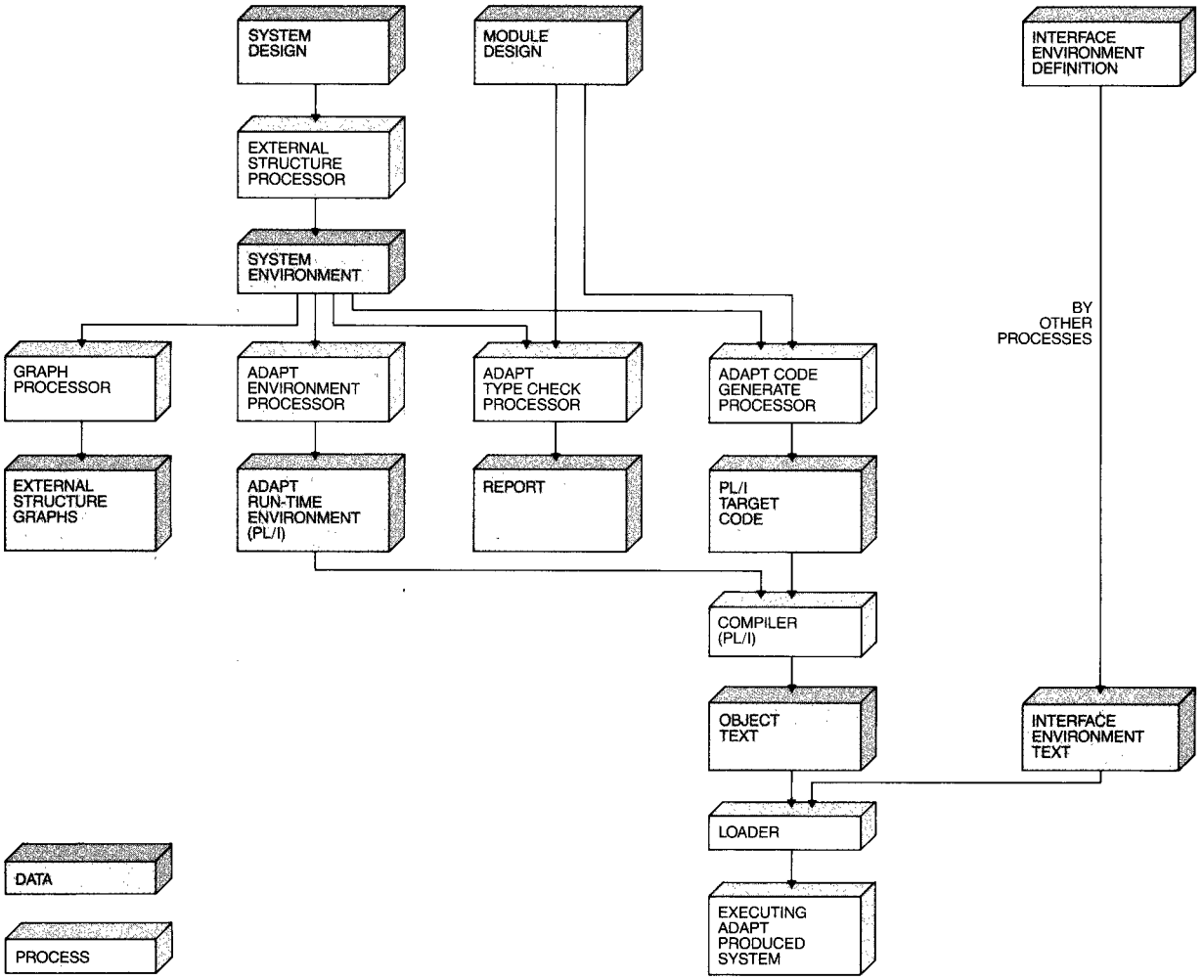
SET is defined as a generic type (capsule) with one type parameter. The EXPORTS list contains the names of the operations that are exported (made visible) outside the capsule. The WHERE clause requires that the type parameter T have an operation EQ that can test the equality of two T objects. The REP declaration defines a RECORD as the internal (concrete) representation of the capsule. This declaration has a SIZE field for the current size of the set and an ELT field for the set elements. ELT is of type FLEX, an unbounded array.

```
  CREATE:
    PROC () RETURNS (*);
      RETURN(REP•CREATE(SIZE:0,ELT:FLEX{T}•CREATE()));
    END CREATE;
```

The INSERT procedure is illustrated by the following example:

```
  INSERT:
    PROC (X:*,Y:T);
      DCL I INT;
      EQUATE F FLEX{T};
      DCL Z F;
      Z = X.ELT;
      I = 1;
```

Figure 1 Components and interrelationships of ADAPT




```
  DO WHILE (I<=X.SIZE);
    IF T•EQ(F•REF(Z,I),Y) THEN
      RETURN;
    I = I+1;
  END;

  X.SIZE = X.SIZE+1;
  CALL F•UPDATE(Z,X.SIZE,Y);

END INSERT;
```

```
IF T•EQ(Z(I),Y) THEN

Z(X.SIZE) = Y;
CALL F•REF(Z,X.SIZE,Y);
```

The first underlines indicate that Z(I) originally appeared in place of F·REF(Z,I) and caused a type diagnostic. The second underlines indicate that Z(X.SIZE)=Y originally appeared and caused the same diagnostic. However, when the statement CALL F·REF(Z,X.SIZE,Y) is substituted, another type diagnostic occurs.

```
DELETE:
  PROC (X:*,Y:T);
    DCL I INT;
    EQUATE F FLEX(T);
    DCL Z F;
    Z = X.ELT;
    I = 1;

    DO WHILE (I<=X.SIZE);
      IF T•EQ(F•REF(Z,I),Y) THEN
        DO;
          CALL F•UPDATE(Z,I,F•REF(Z,X.SIZE));
          X.SIZE = X.SIZE-1;
          RETURN;
        END;
    END;
  END DELETE;
```

In the following example, EACH is defined as an iterator that produces an object of type T. The statement YIELD directs the system to transfer

control to the environment that invokes EACH with the result of the expression following YIELD as the produced object:

```
  EACH:
    ITER (X:*) YIELDS (T);
      DCL I INT;
      EQUATE F FLEX(T);
      DCL Z F;
      Z = X.ELT;
      I = 1;

      DO WHILE (I<=X.SIZE);
        YIELD(F•REF(Z,I));
        I = I+1;
      END;
END EACH;
```

The EMPTY procedure is illustrated by the following example:

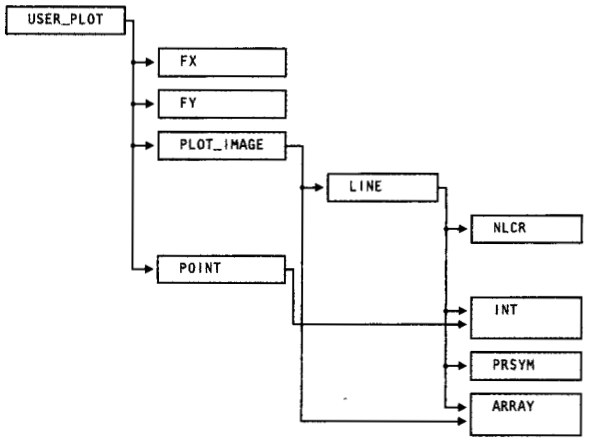
```
  EMPTY:
    PROC (X:*) RETURNS (BOOL);
      RETURN(X.SIZE=0);
    END EMPTY;

END SET;
```

This ends the definition of SET. We now present the module definition for sequence (SEQ) as follows:

```
SEQ:
  CAPSULE (T:TYPE)
    EXPORTS(CREATE,RESET,READ,WRITE,EMPTY);
    EQUATE REP RECORD(PTR:INT,BDY:INT,ELT:FLEX(T) );

  CREATE:
    PROC () RETURNS (*);
      RETURN(REP•CREATE(PTR: 0,
                        BDY: 0,
                        ELT: FLEX(T)•CREATE()));
    END CREATE;

  RESET:
    PROC (X:*);
      X.PTR = 0;
    END RESET;

  READ:
    PROC (X:*) RETURNS (T);                    |    PROC (X:*);
      EQUATE F FLEX(T);                        |
      DCL Z F;                                 |
      IF X.PTR=X.BDY THEN                      |
        CALL ERROR("EOF");                     |
      Z = X.ELT;                               |
      X.PTR = X.PTR+1;                         |
      RETURN(F•REF(Z,X.PTR));                  |    RETURN(X.PTR);
    END READ;                                  |

  WRITE:
    PROC (X:*,Y:T);
      EQUATE F FLEX(T);
      DCL Z F;
      Z = X.ELT;
      X.PTR = X.PTR+1;
      CALL F•UPDATE(Z,X.PTR,Y);
      X.BDY = X.PTR;
    END WRITE;

  EMPTY:
    PROC (X:*) RETURNS (BOOL);
      RETURN(X.BDY=0);
    END EMPTY;

END SEQ;
```

The first underlines indicate that the RETURNS(T) did not appear in the original program and caused a diagnostic. The return type has been stated correctly in the External Structure. The second underlines indicate that RETURN(X.PTR) appears in the original program and has caused a type diagnostic. The READ operation of SEQ in the External Structure calls for a return type of T, whereas the expression X.PTR has the type INT.

Figure 2 Example of an External Structure graph for a small system



System design. In the following section, we show an example of the design of the complete system of a small text editor in ADAPT. We cannot include the definition of the entire editor because the original is about 250 lines. Shown here, however, are one procedure and three data types extracted from this system's External Structure.

```
EDITOR (INSTREAM(STRING),DISPLAY,SCREEN)
  USING
    ( INSTREAM(STRING)    DISPLAY    SCREEN
      EDIT_ENVIRONMENT    FILE       LINE
      COMMAND   ANINTD    FILEID
      NULL      BOOL      INT        CHAR      STRING
      ARRAY(STRING,20)    )

 •••

TYPE FILE
  DEFINES
    ( /* File Manipulation */
      CREATE   (FILEID) -> FILE
      CURRENT  (FILE) -> ONEOF(NORM:LINE,EMPTY:NULL)
      INPUT    (FILE,INSTREAM(STRING)) -> FILE
      INSERT,REPLACE
               (FILE,LINE) -> FILE
      DELETE   (FILE,ONEOF(ANINT:INT,DEFAULT:NULL)) -> FILE
      CHANGE   (FILE,STRING,STRING,
                  ONEOF(ANINT:INT,DEFAULT:NULL),
                  ONEOF(ANINT:INT,DEFAULT:NULL)) -> FILE

      /* Other Operators */
      SETID    (FILE,FILEID) -> FILE
      GETID    (FILE) -> FILEID
      RENUM    (FILE) -> FILE
      TYPEOUT  (FILE,ONEOF(ANINT:INT,DEFAULT:NULL),IO)
```

```
      /* Positioning Operators */
      TOP,BOTTOM,SCROLL,SCROLL_UP
               (FILE) -> FILE
      GOUP,GODOWN,POINT
               (FILE,ONEOF{ANINT:INT,DEFAULT:NULL}) -> FILE
      FIND,LOCATE
               (FILE,STRING) -> FILE  )
  USING
    ( DLIST{LINE}          LINE       FILEID     IO
      INSTREAM{STRING}     ARRAY{STRING,22}
      NULL       INT       BOOL       STRING     CHAR     )
  •••


TYPE LINE
  DEFINES
    ( CREATE        (STRING) -> LINE
      PREFIX,SUFFIX(LINE,STRING) -> LINE
      BEFORE,AFTER (LINE,STRING,STRING) -> LINE
      ALTER         (LINE,CHAR{1},CHAR{1},
                       ONEOF{ANINT:INT,DEFAULT:NULL}) -> LINE
      CHANGE        (LINE,STRING,STRING,
                       ONEOF{ANINT:INT,DEFAULT:NULL}) -> LINE
      ISIN,INITIAL,TERMINAL
                    (LINE,STRING) -> BOOL
      CURRENT       (LINE) -> STRING  )

  USING
    ( NULL       BOOL       INT        STRING     CHAR{1} )

TYPE DLIST {T:TYPE}
  DEFINES
    ( CREATE        () -> DLIST{T}
      BEFORE,AFTER,REPLACE
                    (DLIST{T},T) -> DLIST{T}
      PREV,SUCC,START,FINISH
                    (DLIST{T}) -> DLIST{T}
      ATSTART,ATFINISH
                    (DLIST{T}) -> BOOL
      DELETE        (DLIST{T}) -> DLIST{T}
      OBJ           (DLIST{T}) -> ONEOF{NORM:T,EMPTY:NULL}
      ISEMPTY       (DLIST{T}) -> BOOL     )
  USING
    ( NULL       BOOL )

  •••
```

EDITOR is a procedure with no return value, a main procedure that acts by causing certain side effects. The FILE data type contains operators for CREATE, INPUT, DELETE, and LOCATE among others. The informal semantics of these operators are as follows:

CREATE. Take a FILEID as a parameter and return a FILE, initialized to be empty.

INPUT. Take as parameters a FILE and an INSTREAM (of STRINGs). Successively read STRINGs from the INSTREAM, inserting them into the FILE (at the current location) as encountered. Continue this process until an empty STRING is encountered in the INSTREAM.

DELETE. Take as parameters a FILE and a ONEOF {ANINT:INT,DEFAULT:NULL}, either an integer or the default. If default, delete one line from the FILE at current location. If an INT with value *n*, delete *n* lines from the FILE, starting at the current location.

LOCATE. Take as parameters a FILE and a STRING. Starting with the current location, begin advancing in the FILE until a LINE is encountered that contains the STRING.

In a software production environment, it is appropriate to add the text of these informal semantics to the External Structure source as in-line comments.

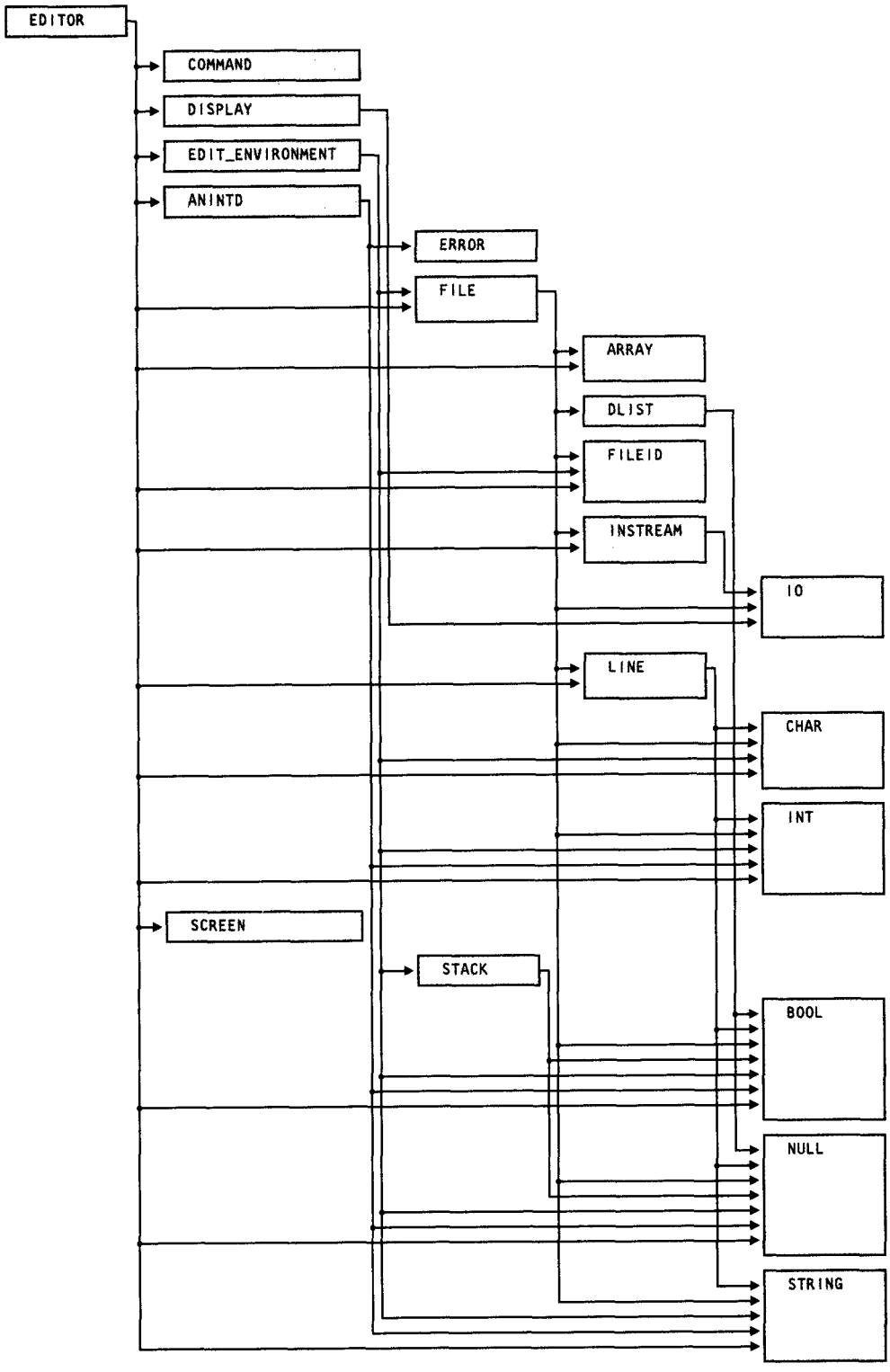
The two other data types defined in this External Structure are LINE and DLIST, which is a generic user-defined data type. LINE is the abstraction for individual lines of the aggregate FILE. As can be seen, operations are included for intra-LINE manipulation, each of which changes a LINE in one way or another. DLIST, a shortening of Doubly Linked List, is an abstraction that is used by the FILE data type. FILE is to have an internal representation that includes a doubly linked list of LINEs, that is, a DLIST{LINE}.

System design involves system decomposition and component and module specification.

Associated with the system description of the EDITOR are the implementation specifications for each module of the system. These specifications are written in the ADAPT language. A little later in this paper we give a portion of the FILE specification.

The USING lists for these modules may seem rather long. When represented graphically, the EDITOR maps into a multiplicity of interconnections, the graph of which we call "bushy." The graph for this system, which is shown in Figure 3, represents the most abstract form of an External Structure. The interface descriptions, both the functional types of procedures and the DEFINES lists for user-defined types, have all been left out. Each module has been reduced to a box. Only the USING lists for each module are displayed. Figure 3 is the External Structure graph for the *entire* editor. The reader can see *all* the required modules of the EDITOR and obtain an idea of what is missing from the description previously given.

Figure 3 Graph of the External Structure of an example editor



```
FILE:
  CAPSULE
    EXPORTS
      ( CREATE,INPUT,INSERT,REPLACE,GETFILE,DELETE,ALTER,
        CHANGE,OVERLAY,CURRENT,FRAME,SETID,GETID,RENUM,TYPEOUT,
        TOP,BOTTOM,SCROLL,SCROLL_UP,GOUP,GODOWN,POINT,FIND,LOCATE );
    EQUATE LINE ONEOF{ANINT:INT,DEFAULT:NULL};
    EQUATE DL   DLIST{LINE};
    EQUATE REP  RECORD{NAME:FILEID,DATA:DL};

  •••

  CREATE:
    PROC (X:FILEID) RETURNS (*);
      RETURN (REP•CREATE(NAME:X,DATA:DL•CREATE()));
    END;

  INPUT:
    PROC (X:FILE,Y:INSTREAM{STRING}) RETURNS (FILE);
      EQUATE INSTR INSTREAM{STRING};
      DCL TSTRING STRING;
      DCL TLINE LINE;
      TSTRING = INSTR•NEXT(Y);
      TLINE = LINE•CREATE(TSTRING);
      DO WHILE (TSTRING¬="");
        X = FILE•INSERT(X,TLINE);
        TSTRING = INSTR•NEXT(Y);
        TLINE = LINE•CREATE(TSTRING);
      END;
      RETURN (X);
    END;

  •••

  DELETE:
    PROC (X:*,Y:ONEOF{ANINT:INT,DEFAULT:NULL}) RETURNS (*);
      DCL COUNT INT;
      SELECT TAG (Y);
        WHEN (ANINT)   COUNT = Y;
        WHEN (DEFAULT) COUNT = 1;
      END;
      DO WHILE (COUNT>0);
        X.DATA = DL•DELETE(X.DATA);
        COUNT = COUNT-1;
      END;
      RETURN (X);
    END;

  •••

  LOCATE:
    PROC (X:FILE,Y:STRING) RETURNS (FILE);
      EQUATE INT_D ONEOF{ANINT:INT,DEFAULT:NULL};
      DCL    TLINE ONEOF{NORM:LINE,EMPTY:NULL};
      DCL    FOUND BOOL;
      DCL    AONE  INT_D;
      TLINE = FILE•CURRENT(X);
      FOUND = FALSE;
      AONE  = INT_D•CREATE(ANINT:1);

      DO UNTIL (FOUND);
        SELECT TAG (TLINE);
          WHEN (NORM)
            IF LINE•ISIN(TLINE,Y) THEN
              FOUND = TRUE;
          WHEN (EMPTY)
            FOUND = TRUE;
        END;
        IF ¬FOUND THEN
          DO;
            X = FILE•GODOWN(X,AONE);
            TLINE = FILE•CURRENT(X);
          END;
      END;
      RETURN (X);
    END;

END;
```

As previously described, the EQUATE for the name REP defines the internal representation for objects of the given user-defined type, and the asterisk (*) is used to refer to a change of interpretation in parameter lists and RETURNS clauses. Note that some FILE procedures do not change interpretation for FILE objects but depend on other FILE operators to do the job.

ADAPT methodology

Software modification is costly in time and resources, and it may also be frustrating and unrewarding to those who do the work. New design approaches can improve this situation. Hiding data representations during design, strong type checking during design, and automated control of system designs are key to ameliorating this issue.

Current practice. System design involves system decomposition and component and module specification. Components are themselves collections of modules. System decomposition today is often an informal process, with the various pieces and components of a system being referred to by name and described using natural language. Interconnections between components are seldom controlled or restricted.

Interface specifications may be recorded in many forms, and the system being built is not generally checked against these definitions. For the most part, the emphasis in this area has been on after-the-fact documentation of module interfaces.

Module specification techniques in current practice include flowcharting, HIPO diagrams, various pseudo-code notations, and a number of structured stepwise refinement strategies.[5,7] Module specification in its most detailed form is commonly called programming and is carried out by use of a *programming language.* Programming languages today allow unsafe data accesses, unchecked module interfaces, and unrestricted intermodule connections or coupling. The use of global variables is a generally accepted practice. A decision made in one module of a software system often has unforeseen and sometimes undesirable consequences in modules far removed from the point where the decision was applied.

Data specifications are normally recorded by giving explicit storage maps, with the result of encouraging premature implementation of system components. This greatly reduces the flexibility needed in an evolving systems design.

In addition, designs can have logical inconsistencies that may not be discovered until late in the development cycle. The most astute and knowledgeable

designer cannot foresee all the consequences of his decisions in a complex system. Discovery of design errors may occur after significant effort has been expended in implementation. The retrofitting of corrections to these errors is often expensive.

Improving current practice. The ADAPT approach improves this picture of design in significant ways. Several improvements occur in both the External Structure and the ADAPT specification language. (1) Executable semantics are provided for designs. (2) Data types may be defined by a user, and (3) generic facilities can improve reusability.

The most astute and knowledgeable designer cannot foresee all the consequences of his decisions in a complex system.

In the area of module specification, there are four primary improvements involved: (1) Incorrect accesses to data are eliminated; (2) Module coupling is limited; (3) Specifications are separated from representations; and (4) Hidden side effects due to global variables are eliminated.

Regarding system specification, four kinds of improvements have also been made: (1) Programming in the large is used for system specification; (2) Compliance of the system specification with module specifications is ensured; (3) A persistent abstract view of a system is promoted; and (4) Separate compilations of modules are handled.

Details on these changes in the design picture are given in the three sections that follow.

Specification improvements in general

Executable semantics for design. ADAPT has been constructed to show designers the consequences of their designs. For those associated with system programming, it seems that the natural way to do this is to execute the designs, that is, to generate the outputs associated with particular inputs. In ADAPT, this is accomplished by translating ADAPT source for the External Structure and module definitions into PL/I code that can be compiled and executed.

User-defined types. Current programming languages provide data types biased toward particular machine architecture, whereas designers should be developing solutions to problems in terms of data types that are relevant to particular applications. Therefore, ADAPT allows designers to specify their own data types.

Full generic capability. A *generic type* is a user-defined type constructor. Thus a generic type is a data type that requires other data types as parameters to complete its definitions. An example of a generic type is a SEQUENCE of elements, where the type of an element is undetermined until the sequence is declared or instantiated. Full generic facilities are provided in ADAPT, not only for data types but also for procedures and iterators.[8]

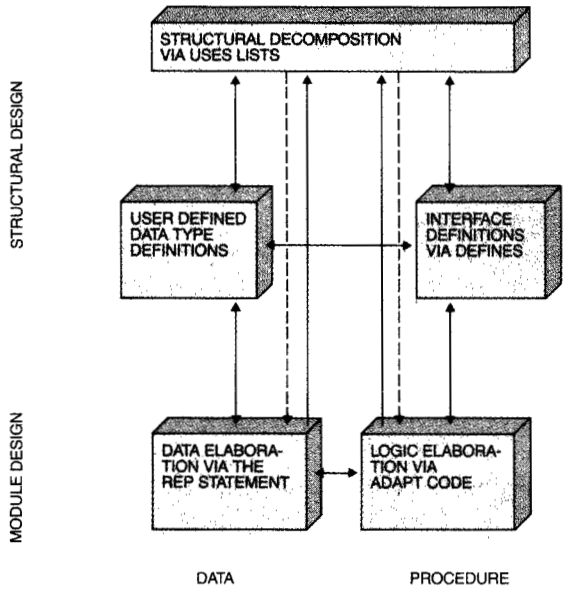
The use of generic facilities supports the production and combination of reusable software components.[9] The generic facilities are appropriate for customizing and tailoring parts from generalized routines.

Module specification improvements

Eliminating incorrect accesses to data. The types of all variables and parameters must be declared in ADAPT. The compiler checks for type agreement between formal and actual parameters in procedure calls, between both sides of an assignment statement, etc. This type checking finds a substantial number of errors at compile time that normally would not be detected until run time.

Changes in module coupling. One of the major problems encountered in the design and development of large systems is that modules interact on global data in unexpected ways. The key to successful program structuring is that of maximizing module independence.[10] This principle is not often followed in the design of large systems, where global control blocks are often referenced by hundreds of modules. Reference to the various kinds of module coupling can be found in Reference 10. ADAPT specifications allow for data-coupled modules. This eliminates undesirable dependencies between otherwise unrelated modules.

Figure 4 Dynamics of the ADAPT design process



Separation of specification from representation. An important principle of system structuring is that of information hiding, a design technique for decomposing systems into highly independent modules.[11] A way of characterizing information hiding is to say that every module hides a design secret, which is usually the format of a particular data structure. The use of data abstraction in the design process causes a strong separation between the specification and implementation of individual modules. The data representation for these modules is hidden within their implementations. Only the specification (allowable accesses) is visible. Changes within a module can be made with minimum impact to other parts of the system.

Restricted side effects. Unrestricted side effects can affect module independence and cause changes to program objects that are difficult to detect. ADAPT permits modules to communicate only by passing parameters. There are no global variables in ADAPT. Side effects can be produced only by updating a data object that is shared as a parameter between modules.

System specification improvements

Programming in the large. ADAPT identifies new steps by which a designer can cause his conception to evolve to a correct design. This evolution is something like the debugging that programmers currently do. In ADAPT, through use of the External Structure, design decisions are expressed in a machinable notation. In such a notation, a designer specifies a system decomposition by defining the system at hand as a set of modules. Module interconnections and interface definitions are added. The designer also describes the activities that each module is to perform. As he expands the design, the designer may change the system structure, thereby altering interconnections or interfaces. At this abstract level of design, the goal is not to produce the best algorithm, but to achieve the right overall structure of the system. Programming in the large for ADAPT involves definition of module decomposition and the interconnections and interfaces for each module.

The notation and tools used for the early stage of design can persist through the life cycle of the system.

Ensuring the compliance of the system specification. System decomposition (programming in the large) is an intellectual activity distinct from module specification, that is, programming in the small or simply programming.[12,13] Since software design involves detailed module specification as well as in-the-large evaluation of system structure, natural support for the design process should reflect this dichotomy of concerns.

In ADAPT, we have a separate language for descriptions in each domain, and we have coupled these descriptions. The ADAPT tools work to keep these two perspectives in correct alignment with each other. Thus we have tools for effectively managing the coupling of a system structure with a set of independent and individually produced modules.

Establishing a persistent abstract view of a system. Designs should be kept up to date. As system

implementation, augmentation, and debugging proceed, fundamental design documentation should be forced into alignment with module programming and changes, as suggested schematically in Figure 4. This process may be compared with the current or traditional process, as suggested in Figure 5. In Figures 4 and 5, the solid lines indicate mandatory update steps, and the broken lines indicate optional update steps.

With the ADAPT package, maintenance of a persistent system description controls and guides this objective. Module descriptions developed within a system are tightly coupled to a system description. The notation and tools used for the early stage of design can persist through the life cycle of the system.

Since the External Structure supplies a persistent view of an entire system, it may be used as a project control facility. It is the repository of design information at different stages in the design process, and can be used to provide system definition support for programmers and development groups.

Separate compilation. The separate compilation of individual modules is a problem that is common to all compilers. This is addressed in ADAPT by use of the External Structure. We compile individual modules with respect to the system description. As long as the interface to a module and its interconnections do not change, the system description can be relied upon to supply all information needed about that module, when compiling other modules.

Experience in using ADAPT

The ADAPT tools are intended to describe and formalize the work that designers do today without tools. The tools capture a description of various activities of system design and the steps of the design process. They provide a terminology and notation for automating system descriptions used in this process. Even so, the use of the ADAPT package requires practices that may be different from the current practices of designers and programmers to whom these tools are new. We have identified certain approaches that describe practices appropriate for using ADAPT.

As in current practice, the decomposition of a system should reflect the ideas around which the designer's innovation is described. This is the keystone of clean design. Modules should be cohesive and limit their coupling to other modules. The addition of data and iterator abstractions extends this notion of good decomposition into new domains. In addition, ADAPT's controlled coupling, strong type checking, and generic facilities change the designer's outlook in important ways by instituting the following requirements:

- A designer should view his system as the collection of modules of which the system is composed. Thus we say that the system has been decomposed into a set of modules or components.
- The interface for a module must be completely defined before the design is complete and testable.
- The interconnections for a system must be defined before the design is complete.
- A system should be constructed with minimal interconnection density. Redundant or irrelevant accesses to modules should be eliminated.
- Cycles in the interconnection logic should be minimized. Cycles should be analyzed carefully; these situations can represent faulty system definition.

Other workers in the field of software engineering[10,14,15] have studied methodologies that govern the

Figure 5 Current or traditional dynamics of the design process

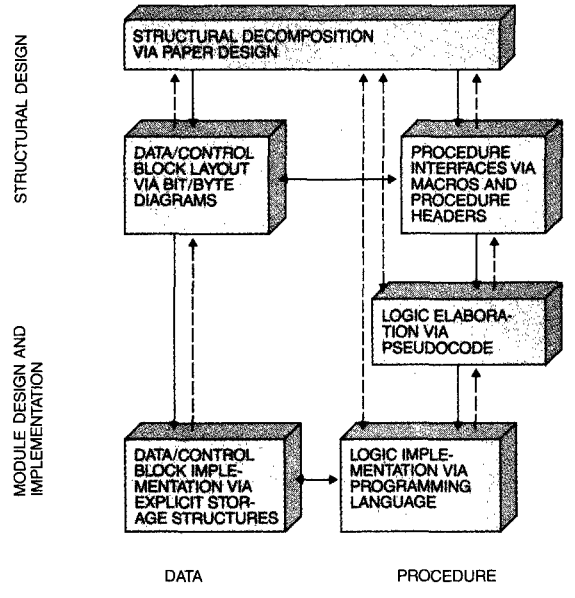
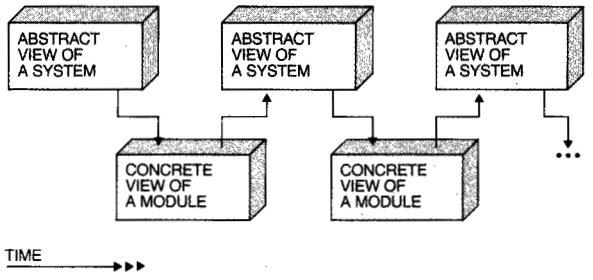


Figure 6 Iterative refinement in ADAPT



construction of the USING relation for a system. Parnas[14] points out that cycles in the USING relation are the source of many difficulties. These situations can represent cyclic definition.[16] There are programs, however, that are naturally described with cycles in the USING relation. In redesign efforts, this has been an inevitable consequence of transliterating data structures where back pointers were used. In our experience, however, we were able to eliminate cycles with a benefit to the design.

Iterative model. In using the iterative model, a user successively refines his definition of a system by shifting his attention alternatively from a high-level abstract notion of his system (correlated with the External Structure) to a low-level concrete view of his system (correlated with the ADAPT language). Figure 6 shows this process. The iterative model also addresses our original research objective of dealing with modifiability. Tracing most modifiability problems to a lack of coordination between a design specification and a semantic specification, the iterative model supplies this coordination. Even very late in the life cycle of a software product, there is an iterative model used in support of product study, upgrades, and modifications.

Experienced users of ADAPT have reported the following approaches to using ADAPT: (1) They maintain two perspectives—the External Structure domain and the ADAPT domain; (2) They avoid preoccupation with either domain, filling in details as needed in the domain being studied; and (3) They alternate frequently between domains. This is our iterative refinement, which occurs as an interaction between the module interconnection language and the module specification language.

Idea-dependency model. Our most abstract experience in using the ADAPT tools has been the idea-dependency model.[16] Considering the design of a system strictly at the first level of decomposition, a solution is decomposed into a set of interrelated ideas, which are either operational or data ideas. For example, in designing an EDITOR, the ideas of FILE, LINE, STRING, and INTERPRETER all seem naturally associated with the concept of EDITOR. First, a user lists all ideas associated with the problem. Then, the internal dependencies of these ideas are indicated. An idea dependency A --> B (i.e., A depends upon B) is indicated whenever the definition of the idea A must make use of knowledge of the idea B. In the EDITOR example, we have FILE --> LINE, because the idea FILE is naturally thought of as an aggregate of the idea LINE.

In our experience with this model, users are quickly able to discuss the potential dependencies of ideas. Something they do not realize at first is how easily these dependencies can form the foundation of a system description. The ideas are mapped into an External Structure as data abstractions, procedures, or iterators. The dependencies, of course, become an initial USING relation.

The ADAPT design process. With our approach, we have been constantly reminded of system development models used by individuals in industry. For example, the simplistic view that design is most naturally a top-down activity does not correspond to the common practice of design that involves extensions or elaborations of an existing system. Here there are natural cycles of incorporating lower-level material into a design. The cyclic nature of the design process is illustrated in Figure 7.

In ADAPT, we have been able to isolate the various tasks that go into projecting a design into a system. Using ADAPT, the steps that coordinate abstract descriptions of a system with a concrete realization are mandatory, as suggested in Figure 4.

The dynamics of these design methodologies are not mutually independent nor do they exclude other possible models. They do, however, provide useful and perhaps new ways to think about programming and systems design. We feel that users will find these descriptions suited to the tools that we have described.

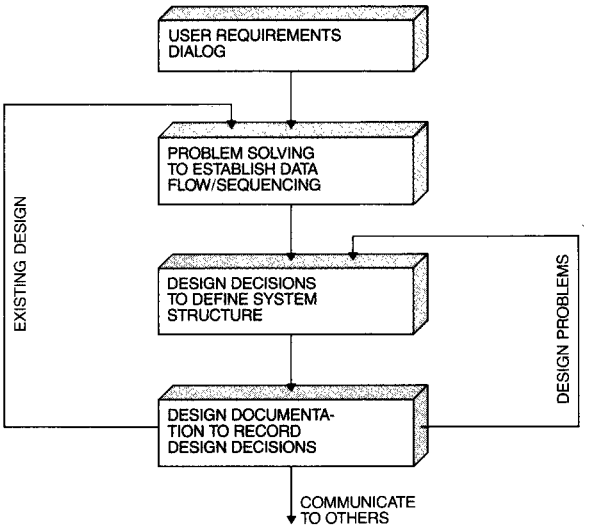
Optimization and related issues

Design tradeoffs. Experience with ADAPT allows us to evaluate certain pragmatic questions. Two areas

of special concern are the following: (1) the performance of generated code, and (2) our ability to integrate this code with programs produced by other processes. Since we have tools for supporting the ADAPT model of software design, we can analyze the advantages and problems that occur.

There is a tradeoff to be acknowledged between the distinct goals of generating high-performance code as opposed to production of system designs that are easy to maintain and can use reusable components. We want to preserve as much of the original design of a system as is reasonable, while adjusting selectively those areas where the greatest improvement is realized.

Figure 7 Cyclic nature of the design process in general



Since ADAPT does not allow global variables, procedure calls could require long parameter lists.

Properties of ADAPT-generated code. By comparison with conventional code, ADAPT code tends to have more procedure calls. Some of these procedures are small and provide trivial accesses to fields of abstract data. In conventional code, these procedures would be explicit in-line references—efficient, but difficult to modify or reuse.

ADAPT data structures usually develop into relatively deep hierarchical structures with a high frequency of indirect references to other hierarchical data structures. By comparison, conventional data structures have a flatter structure (that is, fewer data blocks with more component fields) and require fewer pointer-chasing operations.

Since ADAPT does not allow global variables, procedure calls could require long parameter lists. This is a consequence in current programming languages. However, we have not observed this when using ADAPT. Instead of large collections of independent data items being passed around as parameters, data items tend to become organized into hierarchically structured data types.

Analysis tools. Optimization can be inhibited by a heavy use of procedure calls and indirect data references. What does this cost in overall terms? Our approach is to measure the cost with tools designed for that purpose. Our data are limited, but two things are clear: (1) For many purposes such as design verification and low-demand applications, the use of directly generated ADAPT code is acceptable; and (2) Performance analysis is an effective method for identifying the appropriate sections of a system for optimization. With appropriate analytical tools, we can identify where significant improvement can be accomplished.

Optimization strategy. Current practice encourages the uniform optimization of all code, with little regard to its relative benefits and costs. Preserving the original system design has a value, and this value should be weighed against the benefit of specific optimizations.

In-line expansion of selected procedures can dramatically improve performance. As an example, in one system of 266 procedures, in-line expansion of 13 procedures improved overall system performance by 42 percent.

A key problem with in-line expansion is the tradeoff between performance and maintainability. In-line expansion of an operator of a data abstraction requires exposing, in some way, its otherwise hidden data representation. Future work with representation-hiding methodologies should recognize this and provide tools for the control of such expansions.

Compatibility with existing code. Even though a system can be written entirely in ADAPT, we must also be able to work with an immense body of existing software written in many languages and designed with other methodologies. As these systems are modified and enhanced, we have the opportunity to introduce our new techniques incrementally. To do this requires the interfacing of ADAPT code with other code. This interfacing requires two additional mechanisms in the non-ADAPT code: (1) a more elaborate call facility to accommodate ADAPT environment requirements, and (2) a pseudo-data-abstraction facility to allow accessing existing control blocks. These interface modules are simply convenient step-across modules.

Concluding remarks

The Abstract Design And Program Translator (ADAPT) offers a number of benefits for the construction of reliable software. In the area of abstraction specifically, data abstraction, as well as some types of control abstractions, can be conveniently added to high-level programming with the benefit of increased expressiveness. Extending abstraction structures in this way allows for the specification of entire systems at the abstract level.

The use of automated tools can be propagated in a natural fashion into some of the more abstract areas of software system design. Automating and extending the ideas of programming in the large has enabled us to discuss meaningfully the viewing of systems in their entirety.

Both programming in the large and specification facilities for a broad and useful set of abstractions (e.g., data, procedural, and iterative) can be supported by a common execution facility, thereby allowing designers to test designs as they are being worked on at the conceptual level. The coupling of these two specification areas greatly improves system modifiability.

ADAPT solves the problem of synchronizing design documentation, which is maintained in machine-readable form, with module specifications during the iterative refinement of design and throughout the software life cycle. The External Structure provides this capability by serving as a repository of interface design and system decomposition decisions. The ability to execute specifications during the early stages encourages rapid prototyping and improves design correctness. The use of strong type checking during design improves the system specification process, and many errors are caught at the earliest stages of design. The facility for specification of generic types and procedures encourages the definition of generalized abstractions that can be customized for special cases and permits the construction of reusable software components.

Problems that may be encountered with these approaches include the education of personnel previously trained in other approaches, solving performance problems associated with designs expressed in a very high-level notation, and incorporating systems designed with tools such as ADAPT into an environment which has been developed along other lines. Though these areas have been addressed as part of our research, there is still much work to be done.

Acknowledgments

Many individuals have contributed to the work presented here. In particular, we thank Hamed Ellozy and Laszlo A. Belady, who were previously associated with our project.

Cited references

1. B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction mechanisms in CLU," *Communications of the ACM* **20,** No. 8, 564–576 (August 1977).
2. *Reference Manual for the ADA Programming Language, Proposed Standard Document*, U.S. Department of Defense (July 1980). Also published by Springer-Verlag, New York (1981).
3. J. G. Mitchell, W. Maybury, and R. Sweet, *MESA Language Manual*, Xerox Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304 (April 1979).
4. B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. L. Popek, "Report on the programming language EUCLID," *ACM Sigplan Notices* **12,** No. 2, 1–79 (February 1977).
5. E. W. Dijkstra, "Notes on structured programming," *Structured Programming,* O-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Editors, Academic Press, Inc., New York (1972).

6. R. C. Linger, H. D. Mills, and B. L. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley Publishing Co., Inc., Reading, MA (1979).
7. H. K. Hallman, *A Comparative Study of Pseudocode Notations for Program Design*, Technical Report TR00.3027, IBM Programming Process Group, Poughkeepsie, NY 10602 (November 14, 1979).
8. B. M. Leavenworth, "The use of data abstraction in program design," *Software Development Tools*, W. E. Riddle and R. E. Fairley, Editors, Springer-Verlag, New York (1980).
9. B. M. Leavenworth, *ADAPT: A Tool for the Design of Reusable Software*, Research Report RC9728, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (December 1982).
10. G. J. Myers, *Composite/Structured Design*, Van Nostrand-Reinhold Co., New York (1978).
11. D. L. Parnas, "A technique for software module specification with examples," *Communications of the ACM* **15,** 330–336 (May 1972).
12. F. DeRemer and H. Kron, "Programming in the large versus programming in the small," *IEEE Transactions on Software Engineering* **SE-2,** No. 2, 80–86 (June 1976).
13. C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, John Wiley & Sons, Inc., New York (1982).
14. D. L. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering* **SE-5(2),** No. 3, 128–137 (March 1979).
15. E. Yourdon and L. L. Constantine, *Structured Design*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1979).
16. J. L. Archibald, "The external structure: Experience with an automated module interconnection language," *The Journal of Systems and Software* **2,** No. 2, 147–157 (June 1981).

Jerry L. Archibald *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Archibald joined IBM in 1963, and has been a Research Staff Member since 1970. During this time, he has worked on the design of a number of software systems, including telecommunications (CRBE-CRJE), simulation systems, operating systems, and programming languages (FORTRAN IV). Recent work has been with artificial intelligence systems, including symbolic execution (EFFIGY) and theorem proving systems, as well as software engineering systems (ADAPT) and methodologies. His current interests involve analysis of large-scale views of system definitions (such as those provided by the External Structure), and tools for study of systems in their entirety. After completing his undergraduate education in mathematics and logic at Ohio State University, he did his graduate work at Ohio State, the University of Maryland, and New York University.

Burt Leavenworth *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Leavenworth is a Research Staff Member at the IBM Thomas J. Watson Research Center. Prior to joining IBM in 1961, he was computer center manager at AMF Incorporated and research engineer at Grumman Aerospace Corporation. He was involved with compiler design at the IBM General Products Division, and subsequently worked in the Advanced Systems Development Division on simulation languages. In 1967 Mr. Leavenworth joined the Research Division, where he has been involved with extensible languages, functional programming, and very-high-level languages. For the past five years, he has been working on software development tools based on data abstraction. He received a B.A. in mathematics from Cornell University in 1947 and an M.S. in applied mathematics from Adelphi University in 1956.

Leigh R. Power *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Since joining IBM in 1963 as a member of the Service Bureau Corporation, Mr. Power has designed and implemented systems for I/O control, interactive computing, information retrieval, text processing, and sorting. He has been a Research Staff Member at the IBM Thomas J. Watson Research Center since 1970. There, he is currently working in the field of software technology. In addition to contributing to the development of the ADAPT tools, Mr. Power has become especially interested in the performance implications of data abstractions. Mr. Power received his B.A. in physics from Cornell University in 1963 and attended the IBM Systems Research Institute in 1967 and 1969.

Reprint Order No. G321-5189.