Computer Science Department

TECHNICAL REPORT

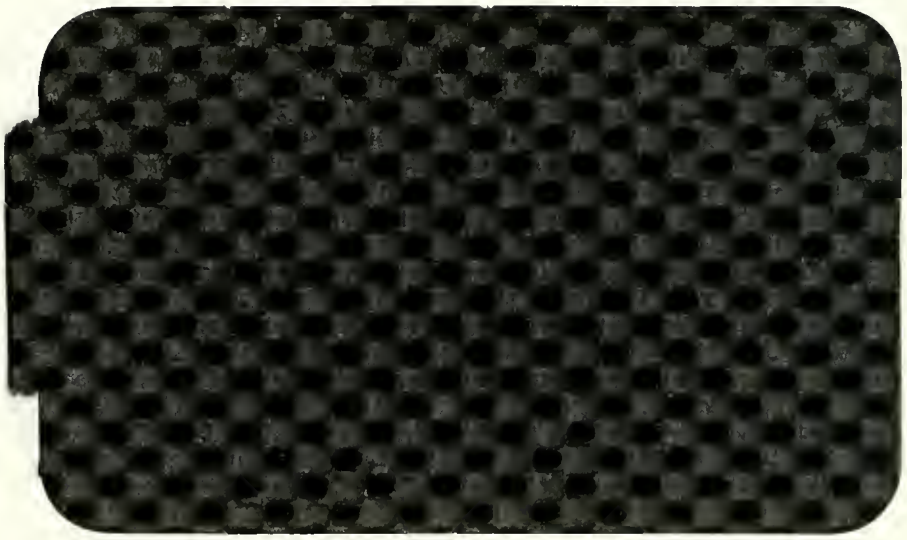
FROM PROTOTYPE TO EFFICIENT IMPLEMENTATION: A CASE STUDY USING SETL AND C.

Edmond Schonberg
David Shields

Technical Report #170
July 1985

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012



FROM PROTOTYPE TO EFFICIENT IMPLEMENTATION: A CASE STUDY USING SETL AND C.

Edmond Schonberg
David Shields

Technical Report #170
July 1985

c.1

FROM PROTOTYPE TO EFFICIENT IMPLEMENTATION: A CASE STUDY USING SETL AND C.

Edmond Schonberg
David Shields
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

ABSTRACT

We report on a large-scale experiment in Software Prototyping using very-high level languages, by describing the evolution of the Ada/Ed system from initial prototype to production-quality translator. Ada/Ed was the first validated translator for Ada, and was written entirely in SETL. A new version of Ada/Ed, written in C, is currently undergoing testing. We discuss the problems associated with a translation from SETL to C, including data structure choices, semantic conflicts between languages, and their impact on the practice of Software Prototyping. We also discuss the promise that wide-spectrum languages hold for Software Technology.

1. Introduction.

This paper reports on an ongoing experiment in the use of very high level languages (VHLLs) for software design and prototyping (see Bu1] for a recent survey of the practice of Software Prototyping). The work of the NYUAda group revolves around the use of SETL, a VHLL whose most salient feature is the use of constructs taken from the mathematical theory of sets. Using SETL, our group was able to construct the first validated translator for Ada® . The first Ada/Ed system, validated in April 1983, was intended to serve as an operational definition of Ada and as a very abstract design for an Ada compiler, and as such was no more than an *executable definition*. (and barely executable at that: a few source lines of Ada per second of CPU time on a VAX/780). Ada/Ed proved nevertheless to be a useful teaching tool, and an excellent testbed for the use of VHLLs in software prototyping; thanks to the exclusive use of SETL, it proved possible to construct a full translator for Ada in about 16 person-years, within a typically unstructured academic environment, without any rigorous Software Engineering procedures, and no design documents other than the program itself [Kr1]. It must be added that roughly half of those 16 years were spent in tracking language changes between preliminary Ada and ANSI Ada, a period in which the prototype did indeed evolve with shifting requirements!

The remarkable inefficiency of the Ada/Ed prototype is due in small part to the inefficiency of SETL itself. More importantly, it is due to the very abstract model that Ada/Ed takes of both the compilation and the run-time environment. In order to show conclusively that Ada/Ed is nevertheless a useful prototype of a production compiler, we have been using it as the design document for a new version of Ada/Ed, written in C, which is to have the performance of an acceptable commercial translator / interpreter. In this paper we report on the construction of the front-end (parser and semantic analyzer) of the new Ada/Ed system, and examine the advantages of using executable prototypes in constructing software systems . This front-end (16,000 lines of C for the parser, 40,000 for the semantic analyzer) represents a sizeable system in its own right, and its completion (as of July 1985) is a good vantage point from which to carry on this examination. A rough chronology of the NYUADA project is given in Fig. 1.

The rest of this paper is organized as follows: section 2 gives a thumbnail sketch of SETL, in order to make subsequent examples intelligible. Section 3 summarizes the evolution of the Ada/Ed system from specification to prototype to production software. Section 4 discusses the central problems which must be addressed in translating a SETL program into C, and the specific solutions that were chosen for Ada/Ed-C. Section 5 describes the performance of the new system. Section 6 briefly discusses some aspects of SETL that hamper rather than facilitate prototyping. Section 7 examines the characteristics of VHLLs such as SETL which contribute to their use as prototyping tools, and concludes with a strong endorsement of the use of wide-spectrum languages for all phases of software design and implementation.

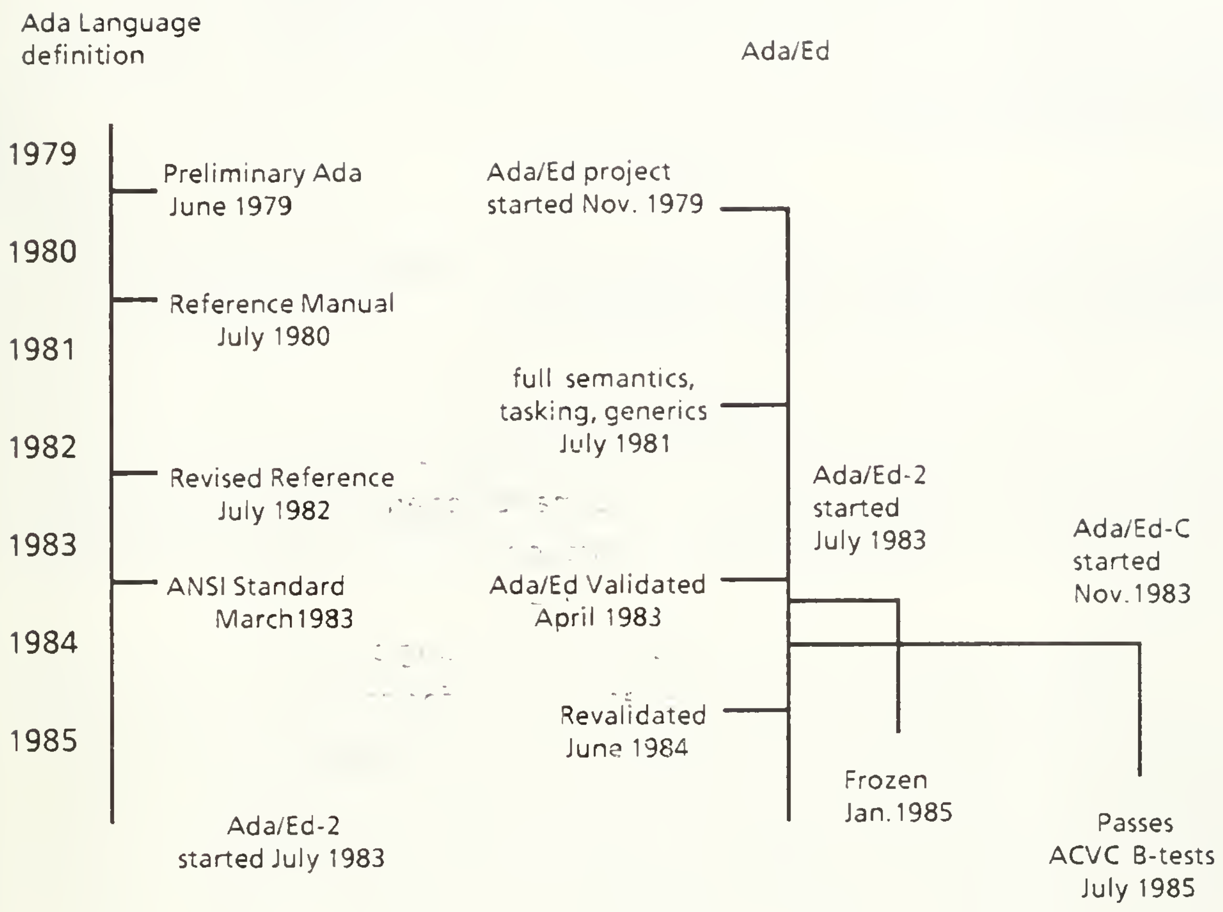

Ada Language definition
Ada/Ed
1979
1980
1981
1982
1983
1984
1985
Preliminary Ada June 1979
Reference Manual July 1980
Revised Reference July 1982
ANSI Standard March1983
Ada/Ed-2 started July 1983
Ada/Ed project started Nov. 1979
full semantics, tasking, generics July 1981
Ada/Ed Validated April 1983
Revalidated June 1984
Ada/Ed-2 started July 1983
Frozen Jan.1985
Ada/Ed-C started Nov.1983
Passes ACVC B-tests July 1985

Fig.1
Chronology of Ada/Ed project.

2. The elements of SETL.

Detailed descriptions of SETL can be found in [De1, De2]. We limit ourselves here to the most salient features of the language, and those whose translation into C presented the more interesting design problems.

SETL is an imperative, sequential language with assignment, weak typing, dynamic storage allocation and the usual atomic types: numeric types, booleans, strings, and generated atoms. Declarations are optional , and typically omitted. The composite types of SETL: **sets, tuples** and **maps**, give the language its expressive power.

a) Sets in SETL have their standard mathematical properties: they are unordered collections of values of arbitrary types, containing no duplicate values, and on which the usual operations are defined: membership, union, intersection, power set construction, etc. Set iterators and constructors are built-in control structures that operate on sets. For example:

{x **in** S | P(x)}

denotes a subset of the set S, all of whose members x satisfy the predicate P(x). Quantified expression over sets use the quantifiers of first-order predicate calculus, and describe common search constructs, For example:

exists x **in** S | P(x)

is a boolean-valued expression whose value is **true** if some member of S does satisfy the predicate P. As a side-effect of its execution, the existentially quantified expression assigns to the *bound variable* x the value of some such member of S, if some exists.

b) Tuples in SETL are ordered sequences of arbitrary size, which are indexed by positive integers. The components of a tuple can be of arbitrary types, so that sets of tuples, tuples of sets, tuples of tuples of sets of integers, etc. can be constructed.. Insertions and deletions from either end allow tuples to be used as stacks and queues; Concatenation makes tuples akin to lists for certain purposes. Heterogeneous tuples (whose components have distinct types) are often used in the manner of the record structures of other languages. Tuple constructors and iterators are similar to the corresponding constructs for sets. For example:

[x: x **in** [2..100] | (**not exists** y **in** [2..x-1] | x **mod** y = 0]

constructs (inefficiently) the sequence of primes smaller than 100.

c) Maps in SETL faithfully reproduce the notion of mapping in set theory: a map is a set of ordered pairs, i.e. tuples of length 2; the first components of these tuples constitute the domain of the map, the second components its range. The elements of the domain and range of a map can be of arbitrary types. Maps can be used as tabular functions, and the *application* M(x) can be evaluated (a map is an associative structure) or be the target of an assignment. Finally, mappings can be multivalued (relations) or single-valued (functions) and the image set M{x} of a value x under the mapping M can also be retrieved and assigned to.

d) Input-ouput is defined for all SETL values, be they atomic or composite. A single command can read or write to a designated file an arbitrary complex structure (set, map, etc.)

These familiar notions constitute the core of SETL. What distinguishes SETL from purely mathematical discourse is the requirement of executability. To guarantee that all valid SETL constructs are executable, only finite objects can be denoted, and they must be built explicitly before they can be used elsewhere (there is no lazy evaluation in SETL). In order to speak about infinite objects in SETL, standard loop and recursive procedure mechanisms are provided, in a conventional syntactic framework akin to that of PASCAL, but without full nesting.

In summary, sets, tuples and maps are *abstract data types* for which the SETL run-time environment has available a full implementation, freeing the user from the need to specify how they should be represented and manipulated. For example, the Ada/Ed interpreter [NY1] describes execution of an Ada program in terms of the state of an abstract mapping (the store) which takes *entities* into their *locations*, and a second map (the contents) which associates *values* with each location. These mappings make no assumptions about an actual memory model (sequential, segmented, etc.) and are thus similar to the constructs used in denotational definitions (and no less abstract than these).

3. The evolution of the Ada/Ed system.

Figure 2 summarizes the successive stages of the Ada/Ed system, from its initial specification to its current status. The box labelled Ada/Ed is the original interpreter, validated in April 1983. The box labelled Ada/Ed-C is the system whose construction is the subject of this paper. The layers in between represent intermediate versions of the prototype, whose purpose was to bridge the semantic gap between SETL and C. It is worth emphasizing here that the conventional distinction between specification and prototype has been blurred in the evolution of Ada/Ed: the description of the executable semantics of Ada in Ada/Ed is certainly no more than a specification, in that a large number of algorithmic details on

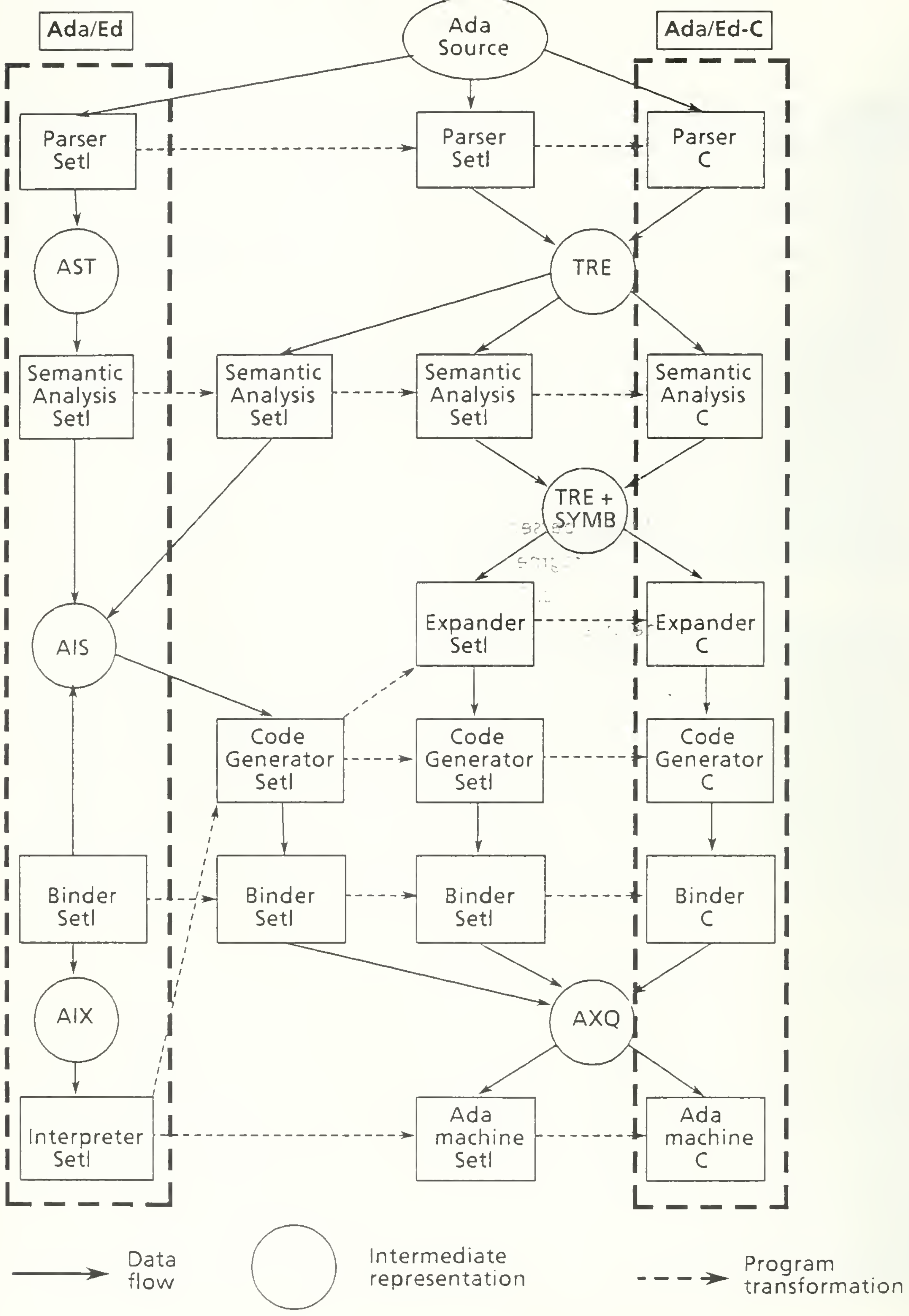


Fig.2 Evolution of the Ada/Ed system

memory and processor management are not explicitly described in it. Ada/Ed (and all its subsequent versions) are nevertheless executable SETL programs, and in fact modules from different versions have coexisted and operated on common program representations, as various data flow paths in the figure show. By the addition of detailed algorithms, additional compiler phases, and the choice of conventional intermediate representations, the initial system has been refined to the point where coding in a conventional systems language is relatively straightforward (and yet far from automatic). This refinement had a different flavor for each of the two main portions of the system. We refer to this low-level SETL version as Ada/Ed-2.

3.1. Evolution of the semantic analyzer.

The original front-end, includng parser and semantic analyzer, is written in a functional style in which tree fragments are constructed and exchanged by procedures with few if any side effects. The only global structure is a simple symbol table with few attributes. The intermediate representation, an abstract syntax tree (AST), is described in SETL as a nested tuple, in a style akin to that of LISP, which in particular implies a large amount of copying . Refinement of the front end consisted in introducing an explicit global tree, whose attributes are modified by procedures with side-effects. The TRE intermediate format is a labelled tree defined by SETL maps. The map retrieval needed to go from node to descendant node thus amounts to an explicit dereference, and brings the SETL text closer to the semantic level of conventional systems languages.

The prototype version uses exclusively the AST as a means of communication between the front-end and the interpreter. This narrow interface is economical for definitional purposes, but unduly restrictive for use by the code generator. As a result, Ada/Ed-2 transmits both TRE and the symbol table to the back-end of the system. The symbol table organization itself is unchanged.

These are the only substantial changes in the evolution of the front-end. The code size is virtally unchanged (around 25,000 lines of SETL) and the algorithms for type checking, overload resolution, generic instantiation and other important aspects of Ada compilation have been kept intact from Ada/Ed to Ada/Ed-2.

3.2 Evolution of the interpreter.

As we have indicated above, the original interpreter is close in spirit to a denotational definition of Ada. As a program, it combines the functions of a code generator and of an interpreter (code generation being described as an activity of the interpreter that builds the continuation to the program, instruction by instruction). This semantic level of description is convenient for a language definition, but is much too abstract to serve directly as a basis for a production compiler. Thus the intermediate version of the interpreter represents a much greater departure from the original than does the front-end. A prototype code generator had to be constructed first, to produce conventional interpretable code for a simple stack machine (the Ada machine). An additional compiler phase was then added (the expander) in order to simplify many aspects of code generation. The three resulting modules: expander, code generator, and Ada machine interpreter, were then ready to be translated into C without additional algorithmic design. It must nevertheless be emphasized that the original Ada/Ed interpreter provided the specification for the compiler phases that subsequently replaced it.

4. Design Issues in translating SETL into C.

The transition from a prototype written in SETL to production software written in a lower-level language consists mainly in chosing concrete representations for the abstract objects appearing in the SETL program, and in tailoring the abstract algorithm therein to these concrete structures. This is typically the province of the programmer *qua* artist, and all of the literature on the efficiency of dynamic search structures can be enlisted in order to choose these representations. Nevertheless, a large number of cases can be covered with a few standard structures of varying complexity. Sets are most frequently represented as hash-tables, as bit-vectors, or as membership bits in property lists; mappings can be encoded as sequences (arrays), if a dense ordering exists on its domain, and so on. We discuss below the more important choices made in Ada/Ed-C.

Apart from bridging the gap between the data structures of the two languages, the passage from prototype to production software must resolve a miscellanea of differences between the semantics of the two languages in the areas of procedure linkage, value vs. pointer semantics, control structures, etc. Here there are few general methods that can be used, and we discuss below what we found most interesting about the use of C as the target language of this translation.

The construction of Ada/Ed-C was influenced by the decision to emphasize readability of the resulting C code, and to try to keep the C code as close as possible to the SETL version. This decision was taken in order to simplify the debugging and testing of the C version, and to simplify subsequent maintenance tasks, The SETL version of Ada/Ed is expected to evolve into a formal definition of Ada, and it will be critical to ensure agreement between the SETL and C versions. Section 5 discusses briefly the efficiency penalties that Ada/Ed-C has incurred because of this decision.

One way to assist readability is to use procedural interfaces to hide the data structure choices and implement set primitives, even in those cases where the structures chosen support efficient set operations that can be coded in-line. By using procedural interfaces, the data structure choices remain flexible, and can be modified once precise measurements are available on the performance of Ada/Ed-C. In this fashion the system still retains some of the flexibility and modifiability of the SETL prototype, and leaves room for further tuning and optimization.

4.1 Structures for sets and iterators.

The following are the important factors in choosing the concrete representation for a set appearing in a SETL program:
a) The operations that apply to it.
b) Its expected size.
c) Whether its representation may safely contain duplicate elements. This last item follows from the fact that in SETL duplicate elements are always removed, which forces a potentially expensive membership test whenever a new element is inserted. If it can be determined that duplicates are harmless, then the set can be described by a *bag*, and implemented efficiently, e.g. as a list. This representation can also be used if it is known a priori that insertions always add new elements to the set.

Examination of the text of Ada/Ed indicates that sets are less frequent than tuples or maps, and typically are small ($<$ 10 elements). Examples of sets include the overload set of subprograms (the set of subprograms with the same identifier that are visible at a given point in the compilation), the set of possible types of an overloaded expression, the set of visible packages in a compilation unit, etc. All of these cases were covered with two representations: sorted arrays, and hash tables. It is interesting to note that the most common set operations in Ada/Ed are membership tests and single insertions and deletions; union and intersection are extremeley rare (less than 20 occurrences in all). This accounts for the fact that no set is represented as a bit-vector.

For sets represented as hash-tables, additional structures are needed to iterate over them. The SETL loop construct:

loop for x **in** S | P(x) **do** ...

is translated by means of a series of iteration macros into the following C fragment:

```
FORSET(x = (Symbol), S, fs1);
    if P(x) {
    ...
    }
ENDFORSET (fs1);
```

In the above, the bound variable x is given an explicit cast (in this case as a Symbol, i.e. a symbol table pointer) in order to allow the macro to be used for sets of different types without triggering spurious errors messages from the **lint** typechecker of the C compiler. The internal variable fs1 is used to hold pointers into the set being iterated upon, and the current value of the bound variable. The ENDFORSET macro does the actual increment and test on the internal pointers that traverse the set.

The implicit iterators appearing in quantified expressions are translated according to similar rules. The common fragment: **if exists** x **in** S | P(x) **then** action(x)... becomes in C:

```
exists = FALSE;
FORSET(x = (Symbol), S, fs1)
    if P(x) {
        exists = TRUE;
        break;
    }
```

```
ENDFORSET(fs1);
if exists {
    action(x)...
}
```

This is a simple but telling example of the conciseness of the SETL dictions, and the clutter that lower-level languages impose on the programmer. Most of the code expansion in going from SETL to C and the corresponding loss of readability are due to such mundane control structures.

4.2 Structures for tuples.

After some experimentation we found the following choice to be the most convenient:

```
typedef char **Tuple;
```

which states that in C a tuple is a pointer to an array of pointers to characters. In practice, not all components of a tuple are pointers to characters, and casts are used wherever needed. SETL tuples are one-origined (the first component has index one) while in C arrays are zero-origined. We found it convenient to use the first component of the C array (the one with index zero) to hold the tuple size. This allows index expressions to be identical in both languages. We note that is somewhat of an abuse of the weak typing of C, where such an object (a dynamic array and its current size, e.g. used as a stack) would more typically be represented by a structure consisting of a size component and an array component. We find nevertheless that the advantage of translating the SETL construct:

```
tup(i) : = val ;
```

into

```
tup[i] = val;
```

rather than into the heavier notation:

```
tup->tup__value [i] = val
```

far outweighs the considerations regarding C style. The same holds for those cases in which SETL uses heterogeneous tuples in the manner of PASCAL records; we use arrays in the C version as well.

The SETL tuple primitives are realized as procedures in C. For example, the code sequence:

```
        tup := [1,3];
        tup with:= x;          $ append x to the end of tup. Now tup = [1, 3, x]
        z fromb tup;           $ assign the first element of tup to z, and delete it.
```

becomes in C:

```
        tup = tup__new(2); tup[1] = (char *) 1; tup[2] = (char *) 3;
        tup = tup__with(tup, (char *)x) ;
        z   = tup__fromb(tup);
```

Procedure tup__new allocates an initial tuple of the specified length using the standard C library procedure malloc. The assignment is necessary in the call to tup__with, as the value of the tuple may be changed because of reallocation.

SETL has copy semantics for assignment, unlike most low-level languages. This means that after:

```
        tup1 := [1, 2] ;
        tup2 := tup1;
        tup1 with:= 3;
```

tup1 has value [1, 2, 3] but tup2 is still [1, 2]. The translation of any destructive operation such as **with:=** thus requires a copy in principle, to avoid side-effects on shared values. In practice, most such destructive uses are safe (values of composites are seldom shared). In the C version, the operation tup__with and its cognates use their first argument destructively, and an explicit procedure tup__copy is used where needed. Note however that only a careful examination of the SETL code will indicate whether a copy is actually needed. The SETL prototype seems here to impose inefficiencies on the production version. This is an area in which the information obtainable from global data-flow analysis may be of considerable help (See [Fr2] for examples).

The converse is that the value semantics of SETL are an important asset to the translation precisely because we know that an assignment is free of side-efects: given that a value in SETL cannot be shared among various objects (there are no pointers in SETL) we know that it is always possble to relocate a composite after it has been modified, because this modification involves only the target of the assignment and is otherwise free of side effects. This is an important simplification in the management of dynamic objects whose size can be expected to vary widely.

4.3 Structures for maps: the parse tree.

As with conventional compilers, the principal data structures manipulated by the front end are the parse tree and the symbol table. Both are represented in SETL as a set of mappings.

Nodes on the parse tree are simply integers, on which several mappings are defined: a kind map, a successor map, whose range type is a tuple of nodes, a value map for terminal nodes (literals and identifiers), a type map for expression nodes, etc.

Parse tree nodes are represented in C as structures. Not all mappings are defined on all nodes. The *kind* of the node functions as a discriminant to interpret the contents of each structure, but the current version allocates the same storage to all nodes, and avoids sharing of storage among maps whose domains are disjoint. The syntactic structure of Ada is such that a node has no more that 4 descendants, or else has a list of them. This allows us to represent the range of the successor map as a tuple of fixed size, or as a list of nodes of the same kind. Other node maps are tuples, sets , and symbols (unique names) which in turn are represented in the C structure by pointers. On the VAX the node structures requires 66 bytes of storage.

The semantic analyzer is a (mostly) top-down pass over the tree, and the most common operation it performs is the unpacking of the descendants of a given node in order to effect semantic checks on them. It is instructive to examine how the code corresponding to a given production (that for an *if*-statement) has evolved in all three versions of Ada/Ed. The tree for this production has the following structure in all three cases:

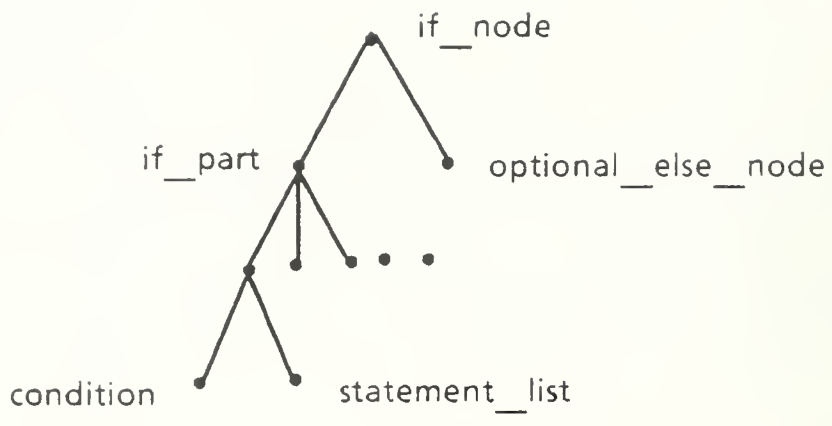


Fig.3 Syntax tree for if__statement.

a) In Ada/Ed, the tree is represented as a nested tuple, and the first component of each tuple is a string that indicates its kind. (in this case, 'if'). At the top level , the semantic processing for this node is expressed as follows:

```
if__part : = [[adasem(cond), adasem(stats)] : [cond, stats] in tree(2)] ;
result : = ['if', if__part, adasem(tree(3))] ;
```

This simply says: process each condition and each corresponding statement list, and then process the final else__clause. In the above, *result* indicates the tree produced by the semantic processing of this node. The semantic processing of the *cond*__node will verify that it is an expression of some boolean type, and so on.(Note the syntax for the iterator over tree(2), which unpacks the pairs into two named constitutents which are bound by the iteration).

b) In Ada/Ed-2, the tree is a global object, and the semantic actions modify and annotate it , rather than rebuilding it. The mapping N__AST, defined over all non-terminal nodes, takes a node into the tuple of its descendants, if there is a fixed number of them. If it is a *list* node, the mapping N__LIST gives instead the sequence of its descendants, all of which have the same kind. The procedure that performs semantic checking of the if__statement now takes the following form:

```
 [if__part, else__node] : = N__AST(if__node) ;
(for node in N__LIST(if__part))
    [cond__node, stat__node] : = N__AST(node) ;
    adasem(cond__node);
    adasem(stat__node) ;
 end for;
 adasem(else__node) ;
```

What makes the semantic level of this fragment lower than the first, is the explicit appearance of the mappings N__AST, N__LIST, whose use is equivalent to an explicit dereference operation. It is characterisitic of the transition to a lower semantic level that storage allocation and pointer mechanisms, which are invisible to the user at the abstract level, must be made manisfest .

c) In Ada/Ed-C, the node is a record whose descendants are accessed through 4 fields called N__AST1,..., N__AST4. If it is a list node, a separate field called N__LIST holds a pointer to a tuple of varying size. The processing of the if__statement now becomes:

```
if__part     = N__AST1(if__node);
else__node = N__AST2(if__node) ;

(FORTUP(node = (Node), N__LIST(if__part), ft1);
    cond__node = N__AST1(node) ;
    stat__node  = N__AST2(node);
    adasem(cond__node);
    adasem(stat__node);
ENDFORTUP(ft1);

adasem(else__node);
```

In this case, the distance between first and second version appears larger than between second and third. In fact the later transformation includes several important data-structuring choices (4 fixed locations for AST descendants, a tuple for N__LIST rather than a conventional linked list) but once these choices have been made (at the global level) the C text can be obtained from the SETL one by following simple coding conventions (which are for the most part embedded in well-chosen macros for control structures).

4.4 Structures for maps: the symbol table.

In Ada/Ed all declared entities are represented internally by unique names, i.e. the equivalent of symbol table pointers. The symbol table proper is a set of maps defined on these unique names. Mappings called Nature, Type, Signature, Scope, Alias, are self-explanatory. A few other mappings are defined for entities of specific kinds. For each entity that may contain other declared entities (subprograms, blocks, packages, tasks, etc.), a separate mapping relates source identifiers to the corresponding unique names. This mapping, called DECLARED, corresponds to the conventional hash-structure used to access the symbol table itself. The structure of DECLARED can be described thus:

```
        DECLARED :  scope__name  → ( id → unique__name)
```

that is to say, DECLARED(scope__name) is a mapping that associates the identifier *id* of any entity declared in *scope__name*, with the *unique__name* (or *symbol)* of that entity.

In Ada/Ed-C symbols are represented by a pointer to a C structure. The range of DECLARED is thus a set of symbols (rather than strings as in the SETL version). Because of the central role played by the DECLARED mapping, several special procedures exist to add a declaration, iterate over the declarations appearing in a scope, search the declarations corresponding to some scope, or to all visible scopes, etc. A typical example is the iteration:

```
        (for [id, u__name] in  DECLARED(some__scope)) ...          end for ;
```

which uses the SETL map iterator, in which the bound variables are an element of the domain of the map, and the corresponding element of the range (maps being equivalent conceptually and structurally to sets of ordered pairs).

In Ada/Ed-C the structure used for a DECLARED map is defined as follows:

```
 typedef struct Declaredmap__s;  {
      short      dmap__length;            /* number of elements */
      short      dmap__hashn;             /* number of hash headers */
      struct Dment **dmap__hasht;         /* pointers to hash table */
 } Declaredmap__s;
```

Each entry in the hash table is a pointer to the following structure:

```
 typedef struct Dment {
    char   *dment__id;                     /* source identifier*/
     unsigned short dment__hash;           /* hash code of identifier*/
    struct Dment  *dment__next;            /*pointer to next entry in  hash list */
    Symbol    dment__symbol;               /* symbol table pointer */
    short dment__visible;                  /* non-zero  for visible identifier*/
 } Dment ;
```

This representation is similar to that used within the SETL system for general sets, except that the types of the domain values (strings) and range values (Symbol table pointers) are known a priori. In the SETL system, hash tables are changed as the size of the underlying set changes; this is not yet done in the C version . The number of hash buckets is thus determined by the initial estimate of the map size given in the call to the procedure dcl__new() that is used to initialize a declared map.

The form used within the C text for iteration over a declared map is the macro

```
    FORDECLARED(id, sym, dmap, dmapiv)
```

which iterates over the declared map *dmap* given as the third argument using the iteration structure *dmapiv* given as the last argument. For each pass through the iteration, the first argument is set to an identifier string, and the second argument is set to the corresponding symbol table pointer, as in the case of the SETL iterator given above. A sample iteration takes the form:

```
    Fordeclared dmapiv ;
    char id;  Symbol sym;
    ...
    FORDECLARED(id, sym, DECLARED(current__scope), dmapiv)
```

The structure used to control the iteration is among the more complicated needed in the translation, and takes the following form:

```
 typedef struct Fordeclared {
        Declaredmap fordeclared__map;
        int fordeclared__i;                    /* iteration index */
        int fordeclared__n;                    /* number of hash headers*/
        struct Dment *fordeclared__p;          /*points to current element*/
        struct Dment *fordeclared__pnext;      /* points to next element*/
        short fordeclared__more;               /* set while entries may exist */
        short fordeclared__vis;                /* set if visible entry */
 } Fordeclared;
```

The macro definitions for iteration over a declared map are then defined as follows:

```
#define FORDECLARED(str,sym,dmap,iv)
        fordeclared__1(dmap,&iv);                      /* initialize iteration */
        while (iv.fordeclared__more) {                 /* while more elements */
           fordeclared__2(&iv);                        /* advance to next one*/
           if (iv.fordeclared-more = = 0) break;       /* quit if done */
           str = iv.fordeclared__p ->dment__id;        / *get identifier*/
           sym = iv.fordeclared__p->dment__symbol;      /*get symbol */
           iv.fordeclared__vis =
                 iv.fordeclared__p->dment__visible;     /*save visibility bit */
           iv.fordeclared__pnext =
                 iv.fordeclared__p -> dment__next;      /* point to next */

 #define ENDFORDECLARED(iv)          }
```

The procedures *fordeclared__1* and *fordeclared__2* are used within the macro to reduce the amount of inline text obtained during macro expansion. The first sets the initial values to start the iteration. The second advances to the next element , either in the current hash chain, or by moving ahead to a non-empty hash chain. The iteration variable fordeclared__more is used to signal the end of the iteration, and is updated in the auxiliary procedure fordeclared__2..

The *Nature* of a symbol functions as a discriminant for the other fields in the structure, but here again the current version simply allocates a fixed size (40 bytes) to all symbols. An additional 28 bytes are added for code generation information.

Other mappings are handled on a case by case basis. Infrequently used maps are represented as tuples of pairs, more common ones as hashed structures, where the hash code of an element of the domain is used to retrieve efficiently the correspondng value of the range.

4.5 External form of intermediate code and separate compilation.

The rules of separate compilation for Ada require that full symbol table information be maintained for separately compiled units to allow for complete semantic checking. The name space of a given unit is affected by other units in the program library, and a unit may make reference to symbols appearing in several other compilations. In SETL, symbols are simply character strings of any length, and no particular distinction need be made between their internal form during compilation, and their external form in a file that holds the result of a given compilation. This simple scheme is not available in C, where symbols are pointers with no a priori external representation. In the C version we choose to represent symbols externally as a pair [file, offset], and to represent the result of a compilation by means of a random-access file. Internally, symbols are also represented by a structure that includes the unit number and sequence number within the unit. The conversion from external to internal form is effected in transparent fashion when a file is read for purpose of separate compilation, and leads to a simple paging scheme for symbol table information.

The single aspect of Ada/Ed where the largest code expansion took place is precisely in the input-ouput of intermediate program representations. As mentioned above, a single SETL statement suffices to perform I/O on arbitrary objects, such as the whole of the DECLARED map. In C such structures must be iterated upon explicitly , and traversed recursively to transform each of their components into or from their external representation. The freedom afforded by the high-level I/O primitives of SETL is one of its greatest assets as a prototyping tool, because it is often the case that in a lower-level language external representations and interfaces are chosen very early in the design, and are prematurely frozen. In the passage from SETL to C, the **last** design choice was that of external representations, in complete reversal of the usual practice, and providing an elegant example of the principle of decision postponement.

5. Some figures.

We estimate that the translation of the semantic phase into C has taken 28 person-months from its inception until the point at which all validation tests of the Ada Compiler validation facility (ACVC) that pertain to semantic checks were successfully passed. The chronology is as follows:

a) One of us (D.S.) began the design of the C version in Fall 1983. Work began with a careful reading of the SETL source and the choice of C coding conventions. The first module to be translated following these coding conventions was a self-contained package for arbitrary precision arithmetic . This initial step took two months and included the time needed to learn C.

b) The data structures for parse tree and symbol table, and the macros for their manipulation were then designed over two months.

c) One month was spent on translating the initialization code (which includes the internal form of the predefined environment and packages of an Ada compilation).

d) Four months were then needed to translate the rest of the code, and three months were spent integrating the system to the point where it could be tested against the existing back-end of Ada/Ed. This required the coding of a first version of the library interface that produced valid input for the SETL version of the code generator.

e) At that point two other members of the NYUADA project took over the completion of the semantic phase including the library interfaces to the C version of the code generator, and the systematic testing using ACVC tests. This process has taken eight months. We expect at least another person-year to be needed for tuning and optimization, for a total investment of under 4 person-years for the production of 40,000 lines of production-quality C-code. This compares to the 5 person-years spent on the 20,000 lines of SETL of the initial initial prototype of the semantic analyzer (about 1/3 of the total spent on Ada/Ed from inception of the project to the first validation), and the 3 person-years needed to produce the intermediate SETL version of the semantic phase. A crude generalization from these figures suggests that roughly equal time be spent on the initial running prototype, the first set of algorithmic transformations, and the production version.

Little attention has been payed to performance so far, and it is only after full ACVC validation of the translator that time will be spent tuning the system. In the meantime, the front-end of the system processes about 1000 lines/min on the VAX/780 under BSD4.2 . This is roughly 25 times faster than the SETL version. We expect that an additional factor of two can be gained by fine-tuning and replacing some procedure calls by in-lne expansions.

6. Weak points of SETL as a prototyping tool.

Some aspects of SETL were the cause for (surmountable) difficulties in the course of translation to C. The most salient of these is the weak typing of SETL: variable are not typed, and can assume values of different types in the course of program execution. There are three situations in which weak typing appears in the program:

a) The programmer has simply reused a name for two unrelated purposes in the same scope. This is both inexcusable and easily remedied.

b) A variable may be weakly typed because it represents a recursive structure (say the parse tree in Ada/Ed) and may correspond to a non-terminal node or a terminal node. Because SETL does not have the equivalent of discriminated unions, the only way to distinguish between them is to use the type predicates **is_tuple, is_string**. This problem disappears in Ada/Ed-2, where discriminants for nodes appear explicitly (but are built by hand, as it were, using SETL mappings, and are not as type safe as discriminants in other languages).

c) A variable may be weakly typed because the problem domain itself suggests several possible types for it. A prime example of this is found in the name resolution routines of Ada/Ed. The notion of overloadable entitites is central to Ada, and refers to the fact that several subprograms or operators with the same identifiers and different parameter profiles may be visible at the same point in the program. When an identifier *id* appears, e.g. in an expression, the name resolution routines examine several scopes (according to the rules of visibility of the language) to determine the meaning of *id*. If it is not an overloadable entity, it has a unique meaning, given by some unique name *u__n*. It may on the other hand be the identifier for several overloadable entities, in which case name resolution yields the *set* of its possible meanings {*u__n1, u__n2, ...*} (it is then up to the overload resolution process to find the single member of this set that is compatible with the context in which *id* appears. See Fig.4).

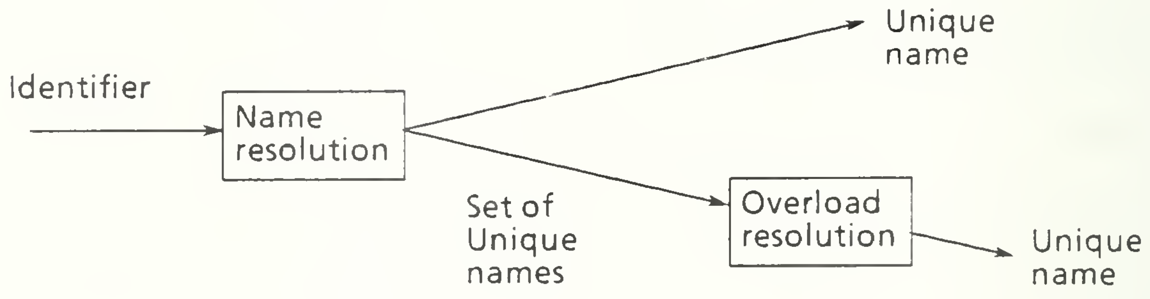


Fig.4 . Name resolution.
Procedure yields either a unique name, or a set of unique names.

In Ada/Ed, the name resolution routine does indeed have these two possible results, and in fact the predicate is__set is used to determine whether a given entity is overloaded. Although this is a faithful paraphrase of the Ada Language Manual, it is awkward to translate into a lower level language, and in fact Ada/Ed-2 chooses a different mechanism, assigning to one of two distinct mappings N__UNQ and N__NAMES, according as the occurrence is overloaded or not. In this case, the weak typing of SETL, which appears at first sight to contribute to a clearer specification of the algorithm, in fact hampers the passage from specification to production, and its use should be avoided. One possible design alternative would have been to return a singleton set for a non-overloadable entity. This was unfortunately precluded by the decision to regard the Ada/Ed predicate *is__overloaded* as synonymous with the SETL predicate **is__set**.

There were other minor instances that may be termed "abuses of the language" in Ada/Ed. In these, SETL itself was not at fault, but instead its power led to programming shortcuts that should be considered unacceptable style. An example relates to the use of strings as unique names. It seemed convenient to construct unique names in the form of expanded names, so that the interpretable code (in which source identifiers are replaced by unique names) should retain some readability, and should have an obvious external representation. Thus, the variable X defined in package P was given the unique name "P.X#18" (the numeric suffix being added to insure uniqueness in the case of overloaded entities). This innocent choice allowed the retrieval of the original name (that is to say X) from the unique name, by means of string manipulations. This usage, intended initially only for error messages, in fact allowed the encoding of additional information in the unique name itself. All such uses had to be removed to produce Ada/Ed-2. In a sense, this indicates a real weakness of SETL, namely the absence of user-defined abstract objects. It also exemplifies a low-level coding trick that should have no place in a software prototype.

7. Conclusions: very-high level languages as tools for prototyping .

The Software methodology exemplified by the Ada/Ed experience is one in which the standard sharp distinction between design and implementation is replaced by a series of refinements applied to successive *executable* versions of a Software system. The first version is conventionally described as a prototype, and its construction is much facilitated by the use of very high level languages. The successive refinements that are brought to it correspond to a lowering of the semantic level of description of the original system, for example by introducing explicit storage management, removing global operations on composite objects, and mainly by choosing concrete representations for abstract data types.

We cannot emphasize enough the extent to which the executability of early versions of the system aids in the construction of subsequent refinements: in the absence of reliable methods for the formal verification of large systems, operational testing remains the best way of ascertaining that the system does what it is intended to do. What distinguishes this use of prototyping from simply "design by debugging" is the use of a VHLL. **If written at a sufficient level of abstraction, the prototype can be as free of implementation biases as any non-executable specification.** Its executability will however simplify considerably the construction of successive refinements. For example, the testing of Ada/Ed-2 and of Ada/Ed-C consisted to a large part in verifying that their internal trace output was the same as that of the Ada/Ed prototype. It must be added that the decision of keeping the C source close to the SETL text has also simplified testing and integration: bugs in Ada/Ed-C were most of the time transcription errors (thus easily detectable) and not conceptual errors. Finally, to the extent that Ada/Ed is free of design errors (it is a validated translator) so is Ada/Ed-C.

After validation of Ada/Ed-C, we intend to continue using Ada/Ed for maintenance. The need for continued maintenance follows from the existence of known semantic gaps in Ada, whose resolution by the Language Maintenance Committee will result in new ACVC tests. Ada/Ed remains a malleable software product, and it will always be easier to modify it first, and then propagate changes to Ada/Ed-C. We thus expect the lifetime of the prototype to be coterminous with that of the production software.

Ada/Ed-2 was also written in SETL, thanks to the fact that SETL can be used at a variety of semantic levels, from the barely executable to the PASCAL-like. SETL is in that sense a *wide-spectrum language* (the term was introduced in [Ba1]). The advantages of programming in such a language are clear: refinements can be applied to critical portions of the system, while less used modules can remain in their original state. In this fashion, an efficient system can be derived from the prototype with a minimum of reprogramming. Ideally, the spectrum of the language should be wide enough to incorporate some machine characterisitics (such as the *register* declaration of C), or at least the semantic level of FORTRAN. This is not the case for SETL, nor any other existing wide-spectrum language, and the Ada/Ed experience indicates that the passage from 'low-level' SETL to C remains a task requiring substantial programming skills. Even though much of this passage resembles standard compilation of high-level control structures, the transformations that we have described are in fact clearly beyond the reach of current optimization techniques. One may hope that purely linguistic mechanisms (data-description languages, libraries of transformations, etc.) could be used to obtain out of low-level SETL code the efficiency of a standard systems language. Work in this area appears promising [De1, Fr1, Sc1] but is still the subject of active research, far from industrial applicability. In the meantime, the use of VHLLs for the first phases of Software production, coupled with the use of prototypes during the lifetime of the software, seems to us the most promising approach for the economical realization of large systems.

Acknowledgements.

This work has been supported in part by US Army contract #DAAB027-82-K-J196 (CORADCOM- Fort Monmouth, N.J). and by ONR contract NO0014856K0413.

We want to express our thanks to Gail Shenker and Bernard Banner, who have completed the translation work on the semantic phase of Ada/Ed-C, and to all the members of the NYUADA group, past and present.

References.

[Ba1] F.Bauer and H. Wiener, *Algorithmic language and Program Developement*, Springer Verlag, Berlin, 1981.

[Bu1] R. Budde et al. *Approaches to Prototyping*, Springer Verlag, Berlin, 1983.

[De1] R. Dewar, A.Grand, S.C. Liu, J.Schwartz and E. Schonberg, "Programming by refinement, as exemplified by the SETL representation sublanguage",*ACM Trans. Program. Lang. Sys.* Vol.1, no.1, July 1979, pp.27-49

[De2] R.Dewar, E. Schonberg and J.Schwartz, *High Level Programming: an introduction to the Programming language SETL,* Counrant Institute of Mathematical Science, New York, 1982.

[Fr1] S. Freudenberger, J.Schwartz and M.Sharir, "Experience with the SETL optimizer", *ACM Trans. Program. Lang. Sys.* Jam. 1983, pp.26-45

[Fr2] S. Freudenberg, *On the Use of Global Optimization Algorithms for the detection of semantic programming errors.* Courant Insitute of Mathematical Sciences Report NSo-24, New York, 1984.

[Kr1] P. Kruchten, E. Schonberg and J. Schwartz, Software Prototyping using the SETL Programming language, *IEEE Software*, Vol.1, No.4, Oct. 1984, pp.66-76.

[NY1] NYUADA group, "An executable Semantic Model of Ada", Courant Institute of Mathematical Sciences, Technical report 84, New York, 1983.

[Sc1] E.Schonberg, J.Schwartz and M.Sharir, "An Automatic technique for Selection of Data Representations in SETL programs", *ACM Trans. Program. Lang. Sys.* Vol.3, no.2, April 1983 pp.126-143.

DATE DUE	BORROWER'S NAME
DEC 19 1989	Frankl