TP 91-07

V

THE DESIGN AND IMPLEMENTATION OF A FOURTH GENERATION PROGRAMMING LANGUAGE

submitted in partial fulfilment of the requirements for the degree

Master of Science

by

Carn Martin Iverson
Department of Computer Science
Rhodes University

ABSTRACT

IV is a very high level language designed for use in a real time production control environment. While most fourth generation languages are intended for use by end users, IV is more suitable for skilled professional programmers.

One of the major design objectives of IV is a dramatic improvement in programmer efficiency during application program development. Non-procedural constructs provided by the language and the use of a number of interactive development tools provide an environment for achieving this goal.

This report presents a language proposal for IV, and addresses related design and implementation issues.

ACKNOWLEDGEMENTS

My thanks to the staff of Steel Information Systems at Middelburg Steel and Alloys, particularly Rob Tennant, Klaus Schwarborn and Marion Bothma, who made this project possible.

Thanks also to my supervisor, Peter Clayton, and colleagues at Rhodes University for their assistance and encouragement this year.

TABLE OF CONTENTS

1. INTRODUCTION

The rapid advance of computer technology is prompting basic changes in the nature of programming. Several factors are responsible for this. Traditional programming languages require skilled programmers of which there has always been a scarcity. Additionally, this demand has led to spiralling labor costs in the data processing industry, compelling corporations to seek alternatives for developing application systems. Demand for application programming work has also increased consistently over the years with existing systems requiring ongoing maintenance, and users requiring the development of new systems as a result of the proliferation of computers. Finally, the system development life cycle approach to software development is no longer adequate to meet users' growing needs in terms of the time taken to develop such systems, the quality of the delivered system, and the degree to which the system conforms to the user's specifications.

Just as the advent of high level languages enabled the programmer to escape from the intricacies of machine code, allowing him to concentrate more on the problem on hand, *higher level programming systems* provide an environment that is easier to use and is more productive than that of preceding software generations. Such tools, collectively called fourth generation environments, shift attention away from the detailed specification of algorithms, towards the expressing and manipulating descriptions of computational processes and the objects on which they are carried out [51].

This report describes one component of such an environment - a fourth generation programming language called IV[1]. IV was developed specifically for Steel Information Systems of Middelburg Steel & Alloys (Pty) Ltd. (MS&A), who run a number of Data General MV20000 super-minicomputers, as an eventual functional replacement of COBOL and FORTRAN for application programming. Initial design specifications required the system to interface to a database management system (DG/DBMS[2]) and an in-house designed screen manager where existing interface techniques yielded numerous programming bottlenecks. Report generation

[1] pronounced I'vee - after the Roman numeral for four

[2] product of Data General Corporation.

would be provided by a separate system called PRESENT[3]. Several other tools provided by MS&A, such as a screen formatter and a query language (IDML[4]), complete the environment.

IV endeavors to promote programmer efficiency, whilst maintaining the run-time efficiency and programming functionality of application programs written in third generation languages to as large an extent as possible. A variety of programming paradigms, including imperative, functional and set theoretic, were considered as a basis of the language. Although functional and set theoretic programming languages provide many powerful, sophisticated abstraction mechanisms, this is often achieved at the expense of machine efficiency. Therefore, IV is primarily imperative, although, like most commercial fourth generation programming languages, several declarative facilities are offered. These facilities are available in the high level interfaces to the database and screen managers, and in tools allowing the programmer to define screen layouts and database components.

The language described in this report represents just the first stage in the life cycle of such a system. It is foreseen that existing constructs will be improved or adapted for changing circumstances, new constructs added to increase the power and sophistication of the language, and new tools added to the environment to promote further productivity increases.

A desirable prerequisite for reading this text is a working knowledge of FORTRAN, DML[5], NATURAL and SQL, as many of the examples of IV's constructs are compared against similar constructs provided by these languages.

Section 2 discusses the problems of defining a fourth generation language as well as some of the concepts that are commonly found in such systems.

---

[3] PRESENT presentation facility - product of Data General Corp.

[4] Interactive Data Manipulation Language - product of Data General Corp.

[5] Database Manipulation Language, defined by the Database Task Group of the Conference on Data System Languages.

Section 3 briefly describes the various components that comprise the IV fourth generation environment.

Section 4 introduces the constructs provided in IV. Much of this work has been presented in a paper entitled *IV: A Fourth Generation Language*, published in the proceedings of the second Conference of M.Sc and Ph.D Research Students [25]. Design aspects of the more interesting constructs are analyzed with reference to similar constructs available in current commercially available fourth generation languages as well as several experimental languages.

Implementation issues are dealt with in section 5, which provides numerous examples of the FORTRAN and DML code[6] generated for particular problems.

[6] Design specifications required that IV statements would compile into FORTRAN 77 code with embedded DML commands.

2. DEFINITIONS AND CONCEPTS

"There is no formal definition of a fourth generation language"

- D.Litwack [40]

"The only characteristics that fourth generation languages have in common is that they are not COBOL"

- S.Gerrard [40]

*"...no single phrase in the software world is as ambiguous and lacking in true meaning as the phrase **fourth generation language**"*

- T.Capers Jones [26]

A decade ago, data processing managers became concerned about their departments' increasing inability to service the computer processing requirements of their users. The software industry's response was to design a wide variety of productivity enhancing tools which were marketed as fourth generation languages. This term was used, and continues to be used, to categorize any new product aimed at improving productivity. As a result there is a great deal of disagreement, aptly described by the above comments, as to what constitutes a fourth generation language.

In order to prevent the continued misuse and abuse of terminology, the following definition, attributed to James Martin, has been adopted by the DP industry: [29,30]

"Any software product used to create applications exhibiting the following two characteristics can be termed a fourth generation language

(1) A ten-fold productivity increase over traditional methods

(2) a knowledgeable user ought to be able to perform useful work after two days training. "

Unfortunately, these characteristics cannot be used to objectively define a fourth generation language.

Firstly, productivity is notoriously difficult to measure. The frequently used *lines-of-code* measurement is clearly inadequate as it does not take into account all the factors influencing productivity. It has also been shown that such measures penalize high-level languages, indicating a reduction in productivity, when productivity has actually increased [26]. In addition, if the reduction in time and effort required to code an application requires a concomitant increase in the utilization of storage and CPU resources, it could be argued that productivity has not increased at all [19].

The second problem is also one of measurement, namely the degree of involvement by the user in application system design. It is argued that not all environments are conducive to end-user computing [6,11,20,24,39], particularly where central database control is required, or where the functions to be coded are complex. An information center, where processing requirements consist of ad hoc reports and queries or simple stand-alone information systems, is an environment where end-user computing is feasible. Most current fourth generation languages are ideally suited for such an environment, as they use powerful commands for information retrieval, hiding virtually all procedural processes. As a result, the involvement of the end-user in application development has been observed as a feature of current fourth generation systems and it has, unfortunately, been proposed as a defining characteristic.

Applications falling in the operational or production environment tend to be rather complex. This complexity requires a degree of formality and procedure not easily expressible in many fourth generation languages. When such languages are used in this environment, the resultant application is difficult to read and maintain. This implies that the involvement of the end user in applications development in this type of environment is limited.

In response to urgent requests, made by the United States Federal government and various private sector organizations, for information and guidance on fourth generation languages, the Institute for Computer Sciences and Technology [12] prepared a report intended to solidify the concept of such languages into an objectively definable entity.

The report, a special publication of the U.S. National Bureau of Standards, proposes a functional model for fourth generation languages. The model serves as a basis for defining terminology, and for describing the services that these languages should offer their users. The report does not claim to be defining a standard, but that it rather presents a framework for possible future standards research [12].

The functional model specifies the capabilities that should be present in a fourth generation language. These capabilities are grouped into three main areas of similar functionality : user functions, data management functions, and system functions [12].

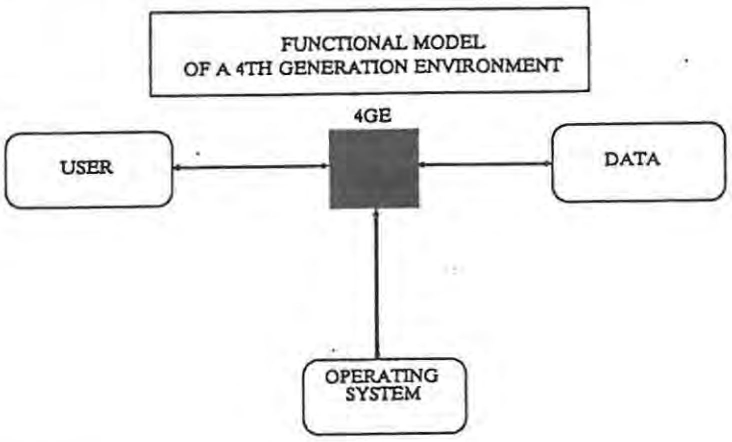


figure 2.1 functional model of a 4GL

User functions are those capabilities necessary to provide a high level dialogue between the language and the users of that language. These users may be human, both technical data processing specialists and non-technical end users, or other application systems. Examples of such functions are screen and menu formatting, screen and menu management, message prompting and logical device management.

Data management functions provide logical data structure management, storage and retrieval of data, archiving and restoration, auditing, and data security. These capabilities are usually provided by an integrated database management system.

System functions provide a means of accessing capabilities not available as part of the language, but which may be provided by the environment in which the language operates. Typical examples of these functions include file handling, job control, and communications.

The term *language* should not be used in the classical programming sense. It is a mechanism for expressing problem solutions, be it in terms of graphics, speech, or written instructions. Therefore, a programming language is but one component of a fourth generation language which may also include an interactive query language, screen generator, report generator, data dictionary, and database management system [5,12,24,29,30]. The functions specified by the model are thus implemented as a system of integrated tools with one or more components providing the services of each of the three major functional areas.

3. OVERVIEW OF IV

IV is essentially a programming language that forms an integral part of an applications development environment. This environment comprises six interacting systems: a data dictionary, a screen manager, a screen formatter, a database management system, a data definition facility, and a very high level language - IV. The interfaces between these components fall into two categories: meta data files, and IV data objects.

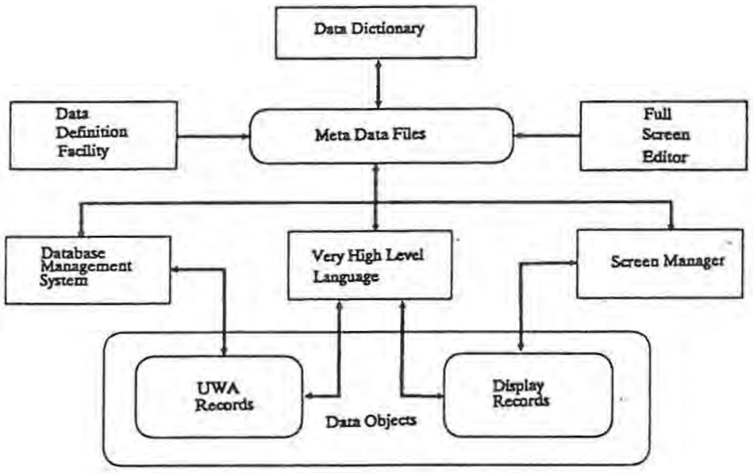


figure 3.1 The IV fourth generation environment

The data dictionary supervises the storage and retrieval of information to and from the meta data files. Meta data files provide definitions of the data objects used by the screen manager, database manager, and IV. Data objects are structures for storing application data. These definitions are described using the screen formatter and data definition facility.

The screen formatter, FSED (Full Screen EDitor [16]), uses full-screen cursor positioning to describe a screen. For each screen that is described, FSED generates two definitions : a screen map containing the physical image that is to be displayed, and a logical definition of that screen containing information such as the co-ordinates, validation criteria, and data entry requirements of each data item.

These definitions are used by the screen manager (FSM - Full Screen Manager [17]) which generates the screen and handles all I/O between the terminal and the IV application. Data is passed between FSM and the application in data objects called display records whose structure is described in the logical definition of the screen.

The Data Definition Facility (DDF) is an interactive, menu-driven tool, which enables a database administrator to specify the structure of a database. The administrator can describe the logical structure of the database by using the Data Definition Language (DDL), and the physical aspects of the database by employing the Data Storage Definition Language (DSDL) [9,39]. The database management system (DBMS) uses these descriptions for the transfer of data between the application and the database. Data is passed between the DBMS and the application in user work area records, data structures which are representations of the database file components that are internal to the application.

A CODASYL[7] compliant network database management system, (DG/DBMS) [9], is employed. Most current fourth generation systems use relational database management systems (such as DG/SQL [10]) which provide the data structure independence needed to make them flexible and easy to use. However, current industry perception is that relational database management systems do not reliably and efficiently handle large numbers of transactions, therefore impeding their use in operational environments where efficiency and security is required. Inverted file (e.g. ADABAS [33,34,38,39,41]), hierarchical (e.g. IBM's IMS [23]) and network database management systems are capable of processing the transaction volumes generated by an operational environment such as the one for which IV was designed.

Access to the data managed by this DBMS is achieved by using Data Manipulation Language (DML [9,15,39]) command statements which allow basic record upkeep operations. These DML statements have to be embedded in a third generation language such as COBOL or FORTRAN 77. A preprocessor is then used to generate the correct calls to the DBMS.

[7] Conference on Data Systems Languages

IV presents high level non-procedural programming constructs for interfacing with the screen manager and the database management system, while retaining programming functionality by the use of Pascal-like procedural constructs and structures, such as procedures, *while* loops and *if* statements. The compiler for IV will translate these constructs into FORTRAN 77 code with embedded DML code for the DG/DBMS interface.

4. THE DESIGN OF IV

One of the major design objectives for IV, was to provide language constructs that facilitate a dramatic increase in programmer efficiency by transferring much of the program development workload from the programmer to the compiler. The resultant increase in the utilization of computer resources is considered acceptable in the light of increasing software costs and decreasing hardware costs [30].

The term, programmer efficiency, is defined as *"the effort involved in coding, testing and maintaining application programs"*. The term is broadly defined and can, for the purposes of this report, be measured according to subjective criteria, so as to avoid the problems of productivity measurement that were discussed in section 2.

Languages providing high degrees of productivity use constructs that exhibit a great deal of abstraction. The greater the degree of abstraction in such a language, the less the amount of detail, or procedural information, that is visible to the user of that language.

Levent Orman's Familial Specification Language (FSL [37]) is one such procedure independent language that influenced initial design considerations for IV. It is a set theoretic and functional language that can be used both as a database language and as an application specification language. FSL allows data independent design of database application systems, providing constructs to accommodate the aggregation and classification problems commonly found in business data processing, and to handle looping implicitly. However, the nature of such constructs requires run-time support that make them unsuitable in operational environments where the machine efficiency of executing applications is of paramount concern.

Several commercially available fourth generation languages were then examined in order to provide less radical constructs. Languages such as Ramis, QBE[8] and ADF II[9], although

[8] Query By Example

providing a great deal of abstraction, do not afford much program functionality. Since an operational environment dictates a large degree of program functionality, the scope of application programs written using these languages are limited. Natural, Ideal and Mantis on the other hand exhibit more proceduralism, thus making them more general purpose. These procedural fourth generation languages therefore serve as a basis for IV. It should be noted that a language such as Natural has been developed over a number of years. Consequently, it is a large language providing a lot of functionality and sophistication. The purpose of IV is not to emulate such a language, but rather to provide constructs which improve on those available in these languages.

An application written in IV comprises a number of modular components, each specifying a set of operations to be performed on a set of data objects.

4.1. IV Application Components

Stevens *et al* [45] discuss considerations and techniques for modularising programs. They point out that dividing programs into smaller pieces, without increasing complexity and hence maintenance effort, is difficult due to the overlapping of code and various other interrelationships that usually exist. These difficulties can be largely overcome by the use of good structured design techniques [45] supported by language constructs that serve to reduce program complexity.

IV program components, called modules (which are similar in syntax and semantics to the Pascal procedure), are pieces of program text which group logically related operations and data objects, and which are referred to by a unique identifier.

The program text consists of two sections: a declaration section followed by an executable section. The executable section specifies the operations to be carried out by a module. Every

9 Application Development Facility Version 2

data object referred to by these operations must be named and declared to be of some fixed type in the declaration section.

A module therefore has the following format:

> **MODULE** *module_name*
> *declaration_section*
> *executable_section*
> **END MODULE**

Each module, which may be compiled separately, has its own set of data objects called program variables. Values for these variables may be passed to other modules via a parameter passing mechanism. The declaration section must specify both the local variables and those that serve as formal parameters.

These intra-application module interactions are termed *module coupling*, a concept that arose from the 1960's structured design revolution [29,30,32,43,44,45]. Several authors [43,45] conclude that a program design objective is to minimize the connections between modules since this will minimize the paths along which changes and errors can propagate to other modules in the system. The widely used technique of using common data areas (required for NATURAL 2 subroutines called with the PERFORM statement[10]) and global variables, can result in an enormous number of connections between modules. The scheme adopted in IV, called *data coupling*, reflects the above goal. The language forces the programmer to clearly specify the data objects to be used by each module, and the interfaces between those modules, thus minimizing possible side effects resulting from erroneous coding.

This scheme supports yet another structured design principle, that of *information hiding* [43,45]. After the function and interface of a module has been completely specified and implemented, the internal details of that module need no longer be considered by the programmer. Therefore, the information required by the programmer for applications development becomes more manageable, resulting in an increase in programmer efficiency.

---

[10] NATURAL Version 2 provides three different modes of subroutine processing [34]. Each mode differs in the way that parameters are passed and the way the subroutine is implemented.

All module calls observe the VS/ECS (Virtual System/External Calling Sequence [15]) convention which allows inter-application module calls and calls to modules written in any of the AOS/VS[11] languages: Pascal, FORTRAN 77, COBOL, BASIC, C, and PL/1. IV also allows other programs to be executed from within the IV application. Here the calling application is suspended until the called program has completed execution.

The above features of IV modules are included in Spector's [42] list of language features that are required to support reusability, which is considered to be one of the solutions to the problem of application backlogs. Many of the other language constructs found in IV, including those powerful commands whose procedural operations are hidden from the user, also fall within the definition of reusability [27,42].

Several organizations have shown that the use of reusable modules greatly increase productivity. Hartford Insurance Company of Connecticut in the United States reported a 7% increase, and programmer productivity rates in excess of twenty thousand source code lines per person year were achieved by Toshiba of Japan [27].

Figure 4.1.1 illustrate the interactions between the various IV application components.

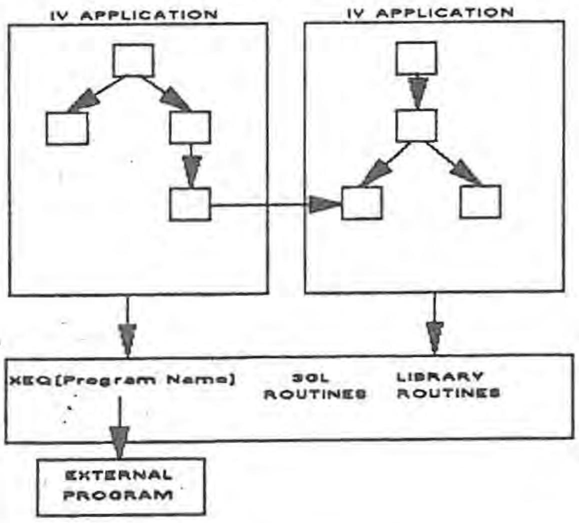


figure 4.1.1 Interaction between IV application components

[11] Advanced Operating System/Virtual System; product of Data General Corp.

4.2. Data Types, Data Objects, Declarations and Scope

Data objects used by programs written in IV contain values of fixed type. The data types are integer, real, logical and character. A data object may also be composed of several values, resulting in a structured type. Structured types include arrays, records and sub-schemas.

4.2.1. Elementary Data Types

Elementary types are based on DG/FORTRAN 77 elementary data types. The following table lists the elementary types available to the IV programmer.

TYPE	DESCRIPTION	STORAGE (bytes)
integer	INTEGER*2	2
	INTEGER*4	4
real	REAL*4	4
	REAL*8	8
logical	LOGICAL	4
character	CHARACTER*n	n

figure 4.2.1.1. IV elementary data types

The binary arithmetic operators exponentiation (**), multiplication (*), division (/), addition (+) and subtraction (-), are available for integer and real operands. The operations of negation (NOT), disjunction (OR) and conjunction (AND) can be applied to operands of type Boolean. Negation is a unary operation while disjunction and conjunction are binary operations. Character values must be enclosed by quotation marks. Non-printable characters are obtained by enclosing their octal ordinal value in angle brackets. For example, "<015>" returns the ASCII carriage return character.

4.2.2. Composed Data Types

Data objects of an elementary data type may be composed into complex data objects of type array, record, or sub-schema.

An array is a sequence of data objects (elements) of an elementary data type. A data object of type array may have up to seven dimensions, the elements of which are stored in column major order. Elements are accessed by the use of FORTRAN subscript notation.

A record is composed of a finite set of fields (named data objects). Each field may be of elementary or array type and may be accessed by specifying the name of that field. Therefore, names given to record fields must be unique. Arrays of records may not be composed.

Sub-schemas are composed of a finite set of logically related record types.

4.2.3 Data Object Declaration

There are three types of data objects: program variables, display records, and user work area (UWA) records. Every data object that is to be used in an application must be named and declared to be of some fixed type.

Program variable declarations are explicitly made within the IV source program text. They follow the FORTRAN format and may be of either elementary or array type. For example, the statements:

```
INTEGER*4 EXECUTIVE_SALARY, TOTAL_SALARIES
INTEGER*2 LITTLE_GREY_MAN_SALARY
```

declare variables of elementary integer type. A variable of array type is declared as above, but with the size of each dimension following the variable name in parentheses. For example:

```
INTEGER*4 A_5_BY_4_MATRIX(5,4).
```

Formal parameter declarations must follow the module name and must be enclosed in parentheses. The example below illustrates a module with parameters A,B and C, and local variables D,E and F.

```
MODULE 100 ( INTEGER*4 A,B
             REAL*8   C )
LOGICAL E
INTEGER*2 D,F
executable_section
END MODULE
```

Display records and UWA records must be named, and are declared of elementary or array data type, with the aid of menu-driven software tools, namely the Full Screen EDitor (FSED) and the Data Definition Facility (DDF). It is in this phase of program development that participation by the end-user is possible. These declarations are made external to the program text, with the definitions stored in a data dictionary. This enables several applications to use the same definitions without having to redeclare those objects.

These tools also allow the programmer to attach attributes to each field of the record. Examples of these attributes include validation processing rules and help files [16], for display record fields, and data protection functions [9,39] for UWA records.

Modules that reference display records, or their components, must name those display records in the declaration section of that module as follows:

```
SCREEN display_record_name, ... , ...
```

If a module references any of the components of a sub-schema, then the following declaration must be included in the declaration section of that module:

```
SUBSCHEMA name_of_sub-schema
```

This declaration makes all the records of that sub-schema visible to the module in which the declaration was made.

4.2.4. Scope

The scope of a data object is that part of the program text in which the data object (or one of its components) may be referenced. The scope is defined by the bounds of the module construct. The scope of any data object is limited to the module in which it has been declared.

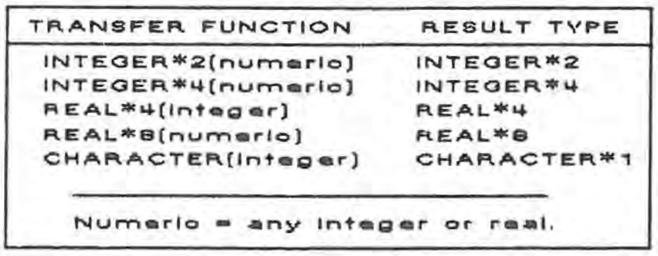
Those program variables declared as formal parameters provide dummy names for variables declared in other modules, so that these variables may be referenced, using the dummy name, outside the module in which they were declared.

The SCREEN and SUBSCHEMA declarations specify that all the components of those display records and sub-schemas that are declared within the program text of a module fall within the scope of that module.

These Pascal-like scope rules, although very simple, provide a powerful mechanism for ensuring reliable application development. This mechanism is arguably superior to that which is available in NATURAL where variable definitions, indicated by using special characters (+,& and #), can occur anywhere in the program text. These characters also determine the scope of those variables [33]. This allows unstructured, undisciplined coding, resulting in increased maintenance effort.

4.3. Expressions

Program variables, numeric literals, UWA record fields, and display record fields may be used in expressions to compute new values. Mixing operands of different elementary data types in an expression is allowed. The types of these operands are converted according to the FORTRAN conversion rules as defined in the Data General FORTRAN 77 reference manual [14]. Programmer defined conversion is specified using type transfer functions as in Modula 2:



TRANSFER FUNCTION	RESULT TYPE
INTEGER*2(numeric)	INTEGER*2
INTEGER*4(numeric)	INTEGER*4
REAL*4(integer)	REAL*4
REAL*8(numeric)	REAL*8
CHARACTER(integer)	CHARACTER*1

Numeric = any integer or real.

figure 4.3.1. IV type transfer functions.

4.4. Statements

A statement specifies one or more operations to be carried out. IV statements fall into either one of two categories, procedural or non-procedural. Proceduralism is defined as the explicit description of an algorithm for the solution of a problem in a finite number of steps. Fischer [12] defines non-proceduralism rigorously as "non-order dependent problem solving". However, no language claiming to be non-procedural strictly satisfies this definition, as the order in which the problem is stated is significant. For instance, database management systems require knowledge of the structure of the database before data can be stored or retrieved.

If we regard a non-procedural statement as an abstraction of a sequence of procedural operations then, limiting Fischer's definition to a single non-procedural statement, the order of these operations need not be specified by the programmer.
These definitions reduce to the following:

- procedural statements specifying *how* something is accomplished.
- Non-proceduralism specifies *what* is to be accomplished without describing *how* it is to be accomplished [6,12,24].

The executable sections of IV modules are composed from non-procedural statements, using procedural statements, nesting and sequencing to construct a module. An analogy can be drawn with constructing expressions, which are composed from numeric literals and variables with arithmetic operators, nesting (by using parentheses), and sequencing of operations serving as expression constructors. Figure 4.4.1 illustrates an IV module composition where the boxes represent the non-procedural statements.

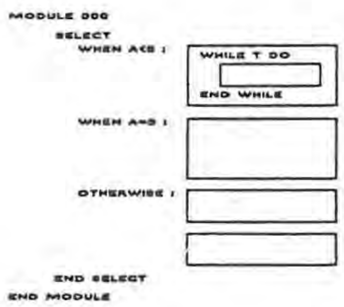


figure 4.4.1 IV module composition.

4.4.1. Non-procedural Statements

The non-procedural statements facilitate high level interaction with the database management system and the screen manager. These statements describe the nature of the interaction rather than the details of how that interaction is accomplished. They also define the operations to be performed on the elements of the UWA and display records.

These statements follow a simple, consistent syntax based on the SELECT...FROM...WHERE format found in SQL [10,39]. This syntax also provides a suitable structure for localizing side effects. Ideally side effects should be avoided completely as in assignment-free applicative languages [21]. However, as explained at the start of section 4, such languages are costly in terms of machine efficiency, making them unsuitable for an operational environment.

An IV non-procedural statement has a mandatory *header clause*, followed by an optional *where clause*, followed by an optional *statement clause*.

4.4.1.1. Screen management

The DISPLAY statement denotes the set of operations required to perform screen management. The *header clause* defines the name of the display record to be used by the screen manager, and determines whether the screen map for that screen record is to be repeatedly displayed. The *where clause* specifies the operations required to initialize the display record's fields before they are displayed. The *statement clause* defines the operations to be performed on those fields after that record has been displayed. These latter two clauses provide a clean interface mechanism to the screen manager. Operations that are pertinent to the screen manager are textually localized, consequently aiding maintenance and improving program stucturation. NATURAL's INPUT...MAP statement, although conceptually similar to the DISPLAY statement described below, does not compel the programmer reflect these logical groupings of operations in the program's structure.

The format of the DISPLAY statement, expressed in EBNF, is as follows:

> **DISPLAY** [**EACH**] [**MENU**] *screen_name* [(*field_for_cursor_position*)]
> *where_clause*
> : *statement_clause*
> **END DISPLAY**

Inclusion of the key word MENU, informs the compiler to generate extra code to handle menu management. To the user menu management appears identical to screen management.

Display records are used to convey data entered by a data capture clerk or a computer operator, to the application program. All terminal I/O is handled by a FORTRAN library routine called FSM. The specification for FSM is as follows: [17]

```
SUBROUTINE FFSM(ISTK,FKY,MODE,NAME,IARR,CARR,RARR,IPOS,IERR)
INTEGER*4 ISTK(*),FKY,MODE,IARR(*),IPOS,IERR
CHARACTER*(*) NAME,CARR(*)
REAL*8 RARR(*)
```

The variable length arrays IARR,CARR and RARR correspond to the display record. The parameter NAME provides FSM with the name of the screen map and its associated display record. The other parameters serve as control variables for FSM (see section 5.2.3.3).

Each screen I/O field is assigned a unique position in one of the arrays IARR,CARR or RARR, depending on that field's data type. Calling FSM from an application written in FORTRAN requires the programmer to assign these positions. For example, a screen with four fields, as illustrated in figure 4.2.2, where the first two fields are of type CHARACTER, the third of type INTEGER, and the fourth of type REAL would require a programmer to remember that they are associated with CARR(1) and CARR(2), IARR(1), and RARR(1) respectively. When the screen has a large number of fields this method becomes cumbersome and can lead to numerous coding errors.

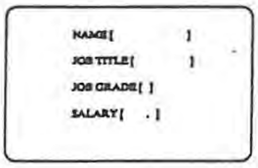


figure 4.4.1.1.1 A_SCREEN screen map - I/O fields are indicated by brackets.

Using the screen formatter, FSED, the programmer can assign a unique name to each screen field. These names can then be used in the program text of an IV application. The compiler will then associate these names to the array elements in the parameter specification of the call to FSM. In addition, the control variables are also managed by the compiler whilst allowing the programmer to access them from within the IV application if so required.

If the screen map illustrated in figure 4.4.1.1.1 is named A_SCREEN and its I/O fields named A,B,C and D respectively, then

```
DISPLAY EACH A_SCREEN ( C )
WHERE A = "N"
: IF A = "Y"
     THEN D = 4.3
          EXIT
  END IF
END DISPLAY
```

will repeatedly display the screen map associated with the A_SCREEN display record. On each display, the cursor will be positioned in the screen field corresponding to C. The variable A- a field of A_SCREEN - is initialized to "N" prior to each display. The screen input/output field corresponding to A will then display a value of "N". The user may type in a new value which is returned by the screen manager to the IV application in A. Execution of the DISPLAY statement will terminate when the screen manager returns the value "Y" in the field corresponding to A, 4.3 in the field corresponding to D, and default values (blanks for character fields and zeros for integer and real fields) in the remaining screen I/O fields. The above statement is thus represented by the following FORTRAN code:

```
10 CARR(1) = "N"
   call SCREEN MANAGER(A_SCREEN)
   if CARR(1) = "Y" go to 20
   go to 10
20 continue
```

The key word EACH, establishes an implicit loop. Omission of the key word EACH, would have resulted in only one call to the screen manager (i.e. the "go to 10" FORTRAN statement above would be omitted).

4.4.1.2. Database management

Christoff asserts that when managing the fourth generation environment "...most attention must be given to (the) creation and care of data, for data is the cornerstone of all fourth generation languages [6]." A language must therefore provide facilities that result in the reliable upkeep of data. These facilities are provided in IV through a set of basic storage and retrieval functions, common to most data manipulation languages, that serve as an interface to the database manager.

Database management statements denote the operations required to move data to and from the database (transfer function), and describe the set of operations to be performed on that data. The *header clause* specifies the transfer function and the target record type. The *where clause* delineates the route through the network to a particular instance of the target record type which is described by a list of conditions (qualification expression). The transfer function can then be applied to the located record instance. Finally, the *statement clause* specifies the operations to be performed on the values of the located record.

Data is transferred between the application program and the network database management system, DG/DBMS, in structures called UWA records. In the network model, data items are grouped together to form record types of which there can be several instances. Record types are associated in pairs by named links, called *sets* and may participate in an unlimited number of such sets, thus forming a network. These associations reflect a parent-child relationship. One instance of a named parent record type may be associated with several instances of its corresponding named child record type, presenting a one-to-many relationship.

Each database record type corresponds to a unique UWA record. As there can be more than one instance of a record type, the UWA record serves as a window onto those instances. That instance of the record type reflected in the UWA record is termed the current record.

The database management statements listed below, specify the transfer of data between the application and the DBMS as well as describing the nature of the interaction with the DBMS.

1. STORE *record_name*
   where_clause
   : statement clause
   END STORE

   creates a new instance of the target record type, storing the current values of the UWA record corresponding to the target record type in it.

2. MODIFY *record_name*
   where_clause
   : statement_clause
   END MODIFY

   copies the current values of the UWA record corresponding to the target record type to the located instance of that type.

3. ERASE *record_name*
   where_clause
   END ERASE

removes the located instance of the target record from the database.

4. GET *record_name*
   where_clause
   : statement_clause
   END GET

copies the values of the located instance of the target record type from the database to its corresponding UWA record.

5. RECONNECT *record_name*
   where_clause
   TO *record_name*
   where_clause
   FROM *record_name*
   where_clause
   use_clause

   CONNECT *record_name*
   where_clause
   TO *record_name*
   where_clause
   use_clause

   DISCONNECT *record_name*
   where_clause
   FROM *record_name*
   where_clause
   use_clause

establishes logical links between located parent and child record types. The use clause indicates into which set the target record type is to be included.

6: READY
   statement_clause
   END READY

Informs the DBMS that a client wishes to make use of the database manager.

7: **AVERAGE, COUNT, MAX, MIN, TOTAL** : are special statistical utility functions with the general syntax:

```
utility_function record_name
where_clause
END utility_function
```

The statement clauses of these data management statements may only specify screen management commands, procedural statements, or simple assignment statements. Thus, the nesting of data management statements is not allowed.

4.4.1.3. Database access

Access to the data stored in a database involves the mapping of the application program's view (external schema) of the database structure, to the logical view (conceptual schema) as discerned by the DBMS.

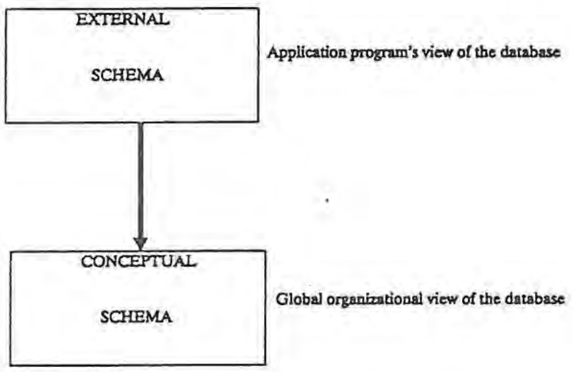

EXTERNAL
SCHEMA
Application program's view of the database
CONCEPTUAL
SCHEMA
Global organizational view of the database

figure 4.4.1.3.1 mapping of external schema to conceptual schema

The relationship between these two views is of paramount importance in the issue of maintenance. Cobb states that "...structure-independent databases are a prerequisite for the successful implementation of fourth generation languages [7]." Structure independence, or logical data independence, insulates the application program from changes made to the conceptual schema. Because of their navigational nature, virtually all non-relational systems fall considerably short of this goal. Pratt & Adamski [39] discuss several relational and non-relational DBMSs' and claim that "while relational systems are better about handling changes, there is still room for improvement [39]." In view of the above assertion, Cobb's statement should be considered a goal rather than a reality.

In the network database model employed by IV, the programmer is required to remember all the path names (sets) in order to navigate from the current position in the network (current root record) to the desired destination. Access to the data is provided by DML, a low level language that operates on single records at a time. Simple relational qualification expressions are used to select the appropriate record. Only one relational operator can be used in a qualification expression. Further relational conditions need to be tested by separate statements, thereby adversely impacting on the expressibility of the application.

Several high level language interfaces for network databases have been developed. An example of such a language is Shneiderman's Pure Data Manipulation Language (PDML [31]). These languages all, however, require the programmer to specify enough set names so that all data items in the qualification expression are connected. Appendix A provides examples of DML, PDML and IV code illustrating record selection methods.

One of the aims of IV is to present a simpler, more usable conceptual view of the network, in which the applications programmer need only know the logical relationships between record types, rather than the physical links associating those types. Full boolean and relational qualification expressions are used for single or multiple record selection thus making database access less restrictive, approaching the relational data model's ease of use.

As mentioned in section 4.4.1.2, the *where clause* of a data management statement specifies a route through the network from an occurrence of the current root record to an occurrence of the record type targeted in the *header clause*. The default root record of any database is a system record type, SYSTEM, of which there is only one occurrence. This record serves as an entry point into the network and cannot be accessed by an application program. Thus, initially, the current root record is set to this instance of the system record type. All navigation through the network starts at the current root record. Consider a database represented by the following Bachmann [9,39] diagram:

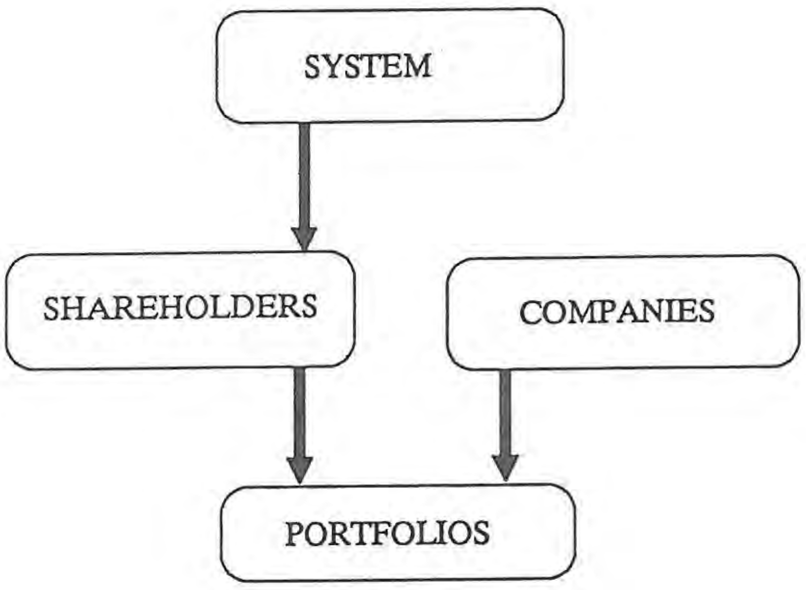


figure 4.3.1.4.1 Logical description of network database

If an instance of the COMPANIES record is to be located, it is necessary to describe a route to that record. This route is as follows: from the SYSTEM record move to an instance of the SHAREHOLDERS record type, then move to an instance of the PORTFOLIOS record type, and finally move to the PORTFOLIOS record parent. For example, the statement

```
GET COMPANIES
WHERE EACH SHAREHOLDERS
      EACH PORTFOLIOS
      OWNER COMPANIES(NAME = "Barlow Rand Corporation")
END GET
```

will retrieve the COMPANIES record for Barlow Rand Corporation. It is necessary to specify that the route to Barlow Rand's record can be via any occurrence (EACH) of the SHAREHOLDER and PORTFOLIOS record types. The *where clause* in the preceding example can be likened to the relational JOIN operation, except that record types (equivalent to tables in the relational model) are not joined on common fields (attributes) but that all combinations of record instances (tuples) are joined until the required record is located or until there are no further combinations. If a record occurrence is not located, a special system variable is set.

Thus, the preceding example could be followed by the following statement:

```
IF NOT FOUND(COMPANIES)
   THEN report_message
END IF
```

to inform the user that no record for Barlow Rand is available.

The current root record can also be changed by the FIND command.

```
FIND SHAREHOLDERS
WHERE SHAREHOLDERS(NAME = "H. Oppenheimer")
: statement clause
END FIND
```

The above statement will set the current root record to H.Oppenheimer's record. This means that the statements in the *statement clause* are restricted to accessing only Mr Oppenheimer's portfolios (PORTFOLIOS) and the companies belonging to him (COMPANIES). Upon execution of the END FIND statement, the previous current root record is restored (i.e. SYSTEM). FIND statements may be nested, thus further restricting access of the database by the *statement clauses* of the nested FIND statements.

Using the DML interface to the database management system, the programmer would have to specify the physical links (sets) required to navigate the database [9,39]. This, together with the simple syntactic and semantic nature of these statements, make complex record location considerably difficult. In IV this task is transferred from the programmer to the compiler. This not only leads to improved programmer efficiency, especially as the programmer need not know the physical structure of the database [7], but the compiler can also generate optimal code for record location. Appendix B provides examples of DML record location and equivalent IV, NATURAL and SQL code.

By preventing the programmer from explicitly naming the sets to be used in database navigation, leaving the compiler to determine which links are to be traversed, IV allows for a degree of logical data independence. This, of course, is not the case when all links between two record types are broken. Access in IV could be improved considerably by using mechanisms similar to IDMS/R's[12] Logical Record Facility and Automated System Facility [39] which present relational-like views of a network database, allowing them to be manipulated as tables using select, project and join operations, and providing improved automatic database navigation.

4.4.1.4. Assignment statements

A common feature of imperative type languages such as NATURAL, is that the scope of variables are determined by a module or some similar construct. This means that the assignment of expression results to an assignment variable may occur anywhere within the bounds of such a construct. Assignments lead to side effects which may adversely affect the debugging and maintenance of application systems. Attempts less radical than those of functional programming languages [21] (which don't allow any explicit assignments) to minimize this problem have been made in SQL and its predecessor, SEQUEL 2 [50]. Update values must be specified by a SET clause which forms part of the UPDATE command. This means that assignments are localized to a piece of program text which is associated to a database management command. However, due to the limited functionality provided by these languages in a stand-alone mode, they have been embedded in third generation languages such as PL/1 and COBOL. As a result of this, the additional scope rule can be circumvented.

IV provides additional scope rules for assignment variables based on those of the stand-alone SEQUEL 2 and SQL languages. These scope rules apply only to assignment variables and not to variables referenced in expressions. All variables, however, are still subject to the scope rules stated in section 4.2.4.

A program variable of either an elementary or array type, may be assigned an expression result by any statement that is within the scope of that variable. If that variable was used as

[12] IDMS developed by the Goodrich Tire Co; acquired by Cullinane Corp in 1971
IDMS/R developed by Cullinet Software, Inc.

an actual argument in a call to a module, then any assignment in the called module to that variable's dummy name assigns the expression result to the actual argument. In the following example, the assignment statement, Y = 1, in MODULE 2 assigns the value 1 to variable X. The same statement in MODULE 1 is, however, illegal as Y is not in MODULE 1's scope.

```
MODULE 1
INTEGER*2 X
 2(X)
 Y = 1
END MODULE


MODULE 2 ( INTEGER*2 Y )
INTEGER*4 X
 X = Y
 Y = 1
END MODULE
```

Fields of a display record may only be assigned values by assignment statements specified in either the *where clause* or the *statement clause* of a screen management statement that references the display record in its *header clause*. For example:

```
MODULE demo
    .
    .
    invalid_a_screen_assignment_statements
    .
    .
  DISPLAY a_screen
    WHERE valid_a_screen_assignment_statements
    :    valid_a_screen_assignment_statements
  END DISPLAY
    .
    .
    invalid_a_screen_assignment_statements
    .
    .
END MODULE
```

A UWA record field may only be assigned values by statements specified in the *statement clause* of a data management statement that references the record in its *header clause* as the target record type. For example:

```
MODULE demo
   .
   .
   invalid_a_record_assignment_statements
   .
   .
 MODIFY a_record
   WHERE locate_an_instance_of_a_record
   :    valid_a_record_assignment_statements
 END DISPLAY
   .
   .
   invalid_a_record_assignment_statements
   .
   .
END MODULE
```

This additional scope rule provides further protection against erroneous coding. Names of fields belonging to records must be prefixed by the names of those records, if referenced by statements falling in the *where clause* or *statement clause* of a non-procedural statement that does not name those records in its header clause. For example:

Consider record A with fields A1 and A2, and record B with fields B1 and B2

```
MODIFY A
WHERE EACH A ( A2 > B.B2 )
: A2 = A1 + B.B1
END MODIFY
```

These additional scope rules provide a significant departure from the freedom afforded the programmer in fourth generation languages such as NATURAL. The resultant flexibility enjoyed by these languages is often at the expense of program maintainability.

4.4.2. Procedural Statements

Many current non-procedural languages cannot be used for serious application development due to limited programming functionality. This problem is overcome by embedding such languages (e.g. SQL,QUEL[13] [39]) in a third generation host language such as COBOL or FORTRAN. Although providing complete programming functionality, this solution results in an awkward two tier programming environment. The programmer is required to differentiate the host language from the embedded language - SQL commands are identified by embracing them with EXEC SQL and END-EXEC statements, whereas QUEL commands are identified by placing '##' on those lines containing a QUEL statement. In IV, as in NATURAL, no distinction is made between procedural commands and commands that result in calls to external systems such as a DBMS or screen manager, thus providing a single uniform programming environment.

These procedural statements are used for explicit looping, conditional branching, and unconditional branching thus affording the application programmer a large degree of functionality. Looping is achieved with the Pascal-like WHILE and REPEAT loops. The semantics of these statements are similar to their Pascal counterparts. As there is no compound statement in IV, an explicit termination statement must be provided for the WHILE loop. This is achieved by delimiting the statements contained in the loop with an END WHILE statement as follows:

```
WHILE a_condition DO
   any_number_of_statements
END WHILE
```

Unconditional branching is achieved by calling either a module or external program, or by using the EXIT statement.

A module call is specified in the same manner as a Pascal procedure call. A module call invokes and passes control to the module named in the program text in the same manner as the Pascal procedure call, except that parameters are always passed by reference. External programs are called by specifying the following statement:

[13] QUEL is the data manipulation language used by the Ingres DBMS

```
XEQ "external_program_name"
```

Control is passed to the next statement upon completion of the external program.

The EXIT statement allows an unconditional branch to the end of the most nested non-procedural statement in which the EXIT statement was defined.

The SELECT and IF statements permit conditional branching. The dangling-else problem is solved by terminating an IF statement with an END IF statement, illustrated by the following example:

```
IF a_condition
   THEN any_number_of_statements
   ELSE any_number_of_statements
END IF
```

The SELECT statement provides a more powerful multiple selection construct than the Pascal case statement. Rather than determining selection on the basis of comparing a single expression against a set of constants, the SELECT statement allows selection according to the results of a set of conditional expressions. The statements associated with the first conditional expression (textually) that evaluates to true, are executed. For example:

```
SELECT
  WHEN FALSE : I = 0
  WHEN TRUE  : I = 1
  WHEN TRUE  : I = 2
END SELECT
```

will always result in variable I having the value one.

4.5. Transactions and Database Consistency

The database may be accessed and modified by several concurrent programs. This can lead to conflicting actions on the same data. "A DBMS must furnish a mechanism to ensure that the database is updated correctly when multiple users are updating the database concurrently" [13,39]. In order to solve these conflicts, transaction boundaries defining transactions are introduced. These boundaries are used by the DBMS to manage concurrent updates.

The format for a transaction is as follows:

```
TRANSACTION
    statements_performing_transaction
END TRANSACTION
```

A transaction is a logical unit of work that performs all the operations of the transaction successfully, or none of them if one is unsuccessful. For example, a typical banking transaction could debit money from one account and credit it to another. If one update occurred without the other, the bank would be either creating or destroying money.

When a transaction starts, the database management system assigns a view of the database to that transaction. A view is a record of all successful transactions completed up to the start of the current transaction. Upon completion of the current transaction, the DBMS checks whether any other transaction has since been completed by comparing the view assigned to the just completed transaction with the current view. If any other transaction was completed, then an update conflict occurred and the DBMS returns an error code to the application. The longer the transaction (in terms of the number of operations to be performed), the greater the likelihood of update conflicts arising.

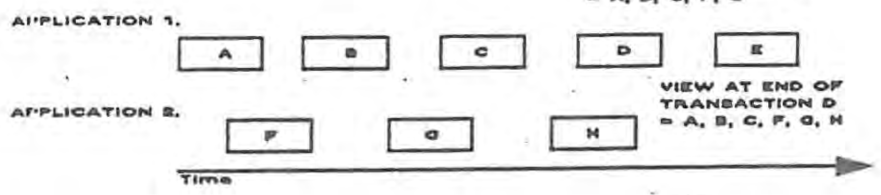


figure 4.5.1 Illustration of database views

The construct outlined above can be used to clearly demarcate the start and end of a logical unit of work. It provides greater program clarity than either SQL[14] or NATURAL[15], and in the case of SQL, provides greater efficiency as well.

In SQL the start of the program marks the start of the first transaction. The COMMIT command indicates the end of that transaction and the start of the next one. The programmer, contrary to the above definition of a transaction, may be required to include unrelated operations in a transaction since no piece of program text can fall outside a transaction boundary. Therefore, this construct must also have an adverse impact on efficiency, particularly in a large multiple user environment.

NATURAL does not provide an explicit start-of-transaction command. The first database management command following an END TRANSACTION command, or the start of a module, designates the start of a transaction with an ensuing END TRANSACTION command denoting the end of that transaction. Although probably as efficient as IV, the lack of an explicit start-of-transaction statement restricts the clarity and stucturation of the application program.

4.6. Error handling

Cocco and Dulli [8] discuss language features for dealing with exception conditions. In keeping with the tenets of structured programming and the design objectives of IV, such features should clearly distinguish between normal control flow and the exception handler.

IV provides a structured construct for dealing with errors generated by non-procedural statements, based on the Ada exception handler construct. It allows the programmer to deal with *escape, notify* and *signal* exceptions as classified by Goodenough [18].

---

[14] Using IBM's DB2 DBMS running under MVS/370 or MVS/XA

[15] Using The ADABAS DBMS

An error generated by a non-procedural statement results in control being passed to an error-handler for that statement. Error-handlers are statements defined immediately prior to the END statement of either a non-procedural statement, or a transaction statement. If the error is not dealt with immediately, control passes to the error-handler of the next outermost non-procedural or transaction statement. This is illustrated by the following example:

```
MODIFY EMPLOYEE
WHERE FIRST DEPT(NO = 301)
: SALARY = SALARY * 1.16
  WHEN DBSTATUS = some_integer : do_something
END MODIFY
```

If DBMS should return an error when modifying the employee record, then control passes to the WHEN statements declared prior to the END statement of the MODIFY command. If the value of DBSTATUS corresponds to *some_integer*, then *do_something* is executed instead of the previous statements defined in the *statement clause* of the MODIFY command. If DBSTATUS is not equal to *some-integer* then control will pass to the WHEN statements of the command in which this MODIFY was defined, as shown below:

```
DISPLAY S01
: perform_modification_specified_above
   some_other_statements
   WHEN DBSTATUS = another_integer : do_something_else
END DISPLAY
```

More than one WHEN statement may be defined in a *statement clause* of either a non-procedural or transaction statement, provided they are consecutive statements. The WHEN statements form alternatives to the other statements in a *statement clause*. After control has been passed to them, execution proceeds normally. The following example code text illustrates this point.

```
MODULE 000

INTEGER*2 FLAG

REPEAT
  TRANSACTION
    FLAG = 0
    DISPLAY EACH S01
    WHERE some-initial-statements
    : IF QUIT = "Y" THEN EXIT
      perform-modification
    END DISPLAY
    WHEN DBSTATUS = 17202 : MESSAGE = "UPDATE CONFLICT-REENTER DATA"
                                        FLAG = 1
                                        EXIT
  END TRANSACTION
UNTIL FLAG = 0

END MODULE
```

If the EXIT statement defined in the WHEN statement had been omitted, the END TRANSACTION would have been executed. Since this statement results in a COMMIT DML command being executed, another error would have been generated because the initial error has not been cleared. In order to clear an error it is necessary to restart the transaction. The EXIT statement would cause control to pass to the statement immediately following the END TRANSACTION statement. Since FLAG has been set to one, the statements in the REPEAT loop will be executed again, thus restarting the transaction.

4.7. Comments

All comments in IV, which may be located anywhere in the program text, are enclosed by "{" and "}". For example:

```
{ this is an example of a comment }
```

Comments may not be embedded in other comments.

5. IMPLEMENTATION

IV was implemented for a DP department as a commercially viable system. The situation, unfortunately, provided little opportunity for research into aspects of implementation, resulting in standard, well-known techniques of implementation being employed. This section, rather than discuss what is already familiar to the reader, concentrates on presenting several interesting aspects of FORTRAN and DML code generation. A conceptual description of the compiler is also provided, as well as a brief evaluation of the languages that were available for implementing the compiler (AOS/VS COBOL[16], AOS/VS FORTRAN 77[17], AOS/VS Pascal[18]).

The choice of a programming language to implement a system must be based on criteria which reflect the fundamental concepts of good systems design as well as the language's suitability for the particular problem to be solved [2,28].

Modularity is one of the most important criteria required for programming in the large [28]. To satisfy this criterion, it must be possible to develop independent sub-systems and compile them separately. Facilities for developing subroutines are poor in COBOL, making program modularity almost impossible. Consequently, this language received no further consideration. Both AOS/VS FORTRAN 77 and AOS/VS Pascal provide recursive sub-program facilities. However, these facilities are more limited in FORTRAN - parameters are only passed by reference and sub-programs may not be nested. Separate compilation is also provided by both of these languages. A module construct similar to that provided in Modula-2, is supported in AOS/VS Pascal. Explicit export and import of variables, procedures and functions make this facility far more powerful than the FORTRAN facility [14,15].

---

16 based on ANSI Standard Cobol (X3.23-1974)

17 conforms to ANSI Standard FORTRAN (X3.9-1978)

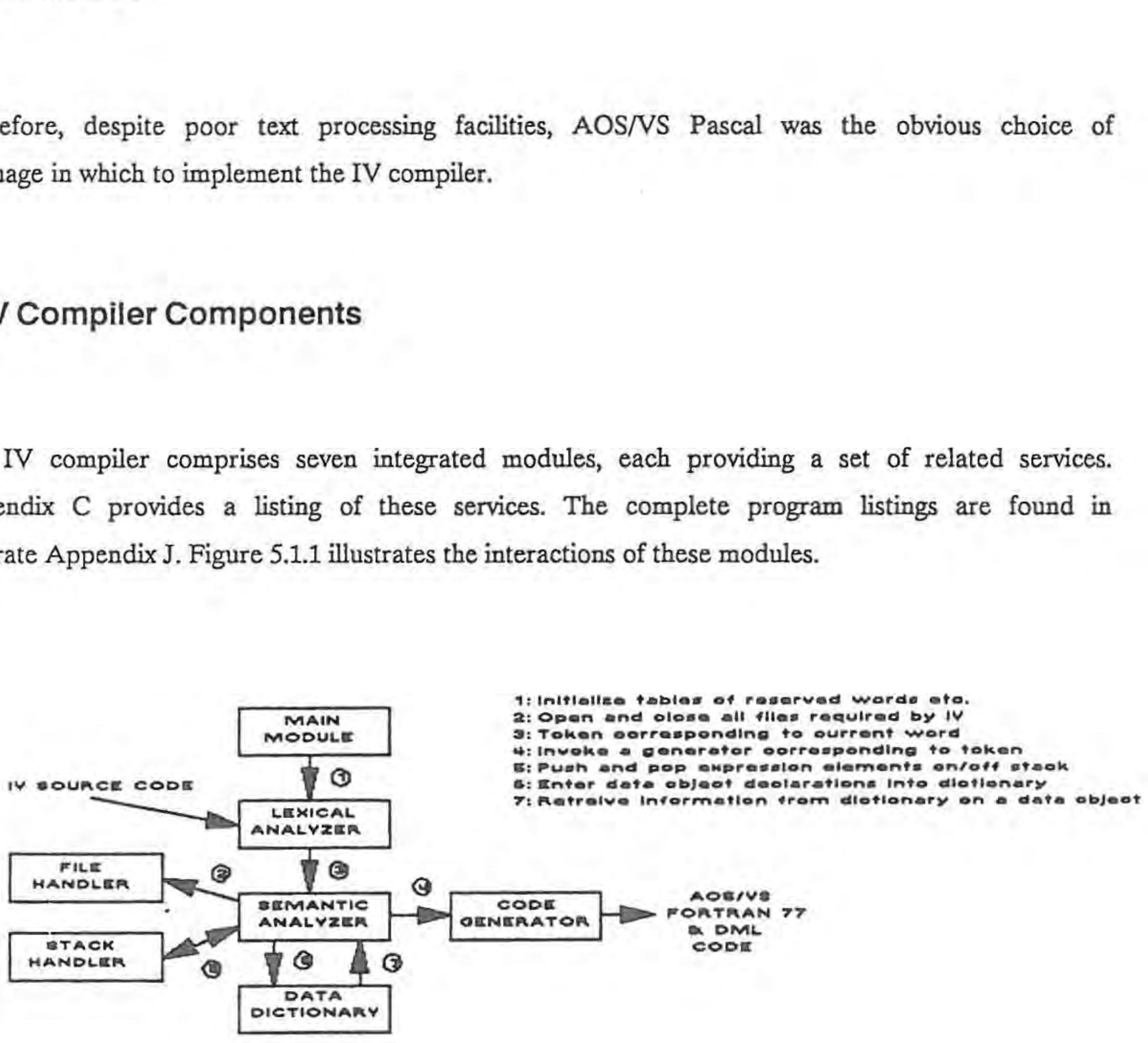
18 conforms to Draft Proposed American National Standard Pascal(ANSI X3J9-IEEE)

Another important criterion is that of reliability. Although the reliability of a system is highly dependent on the programmers of that system, it has been shown [28,29,30] that the use of structured programming languages have a considerable impact on programmer reliability[19]. A related issue is that of a language's ability to handle modifications and extensions. This also requires modularity, clarity and structuration [28,45,49,52,54]. All of these features are well supported in Pascal. FORTRAN 77, although introducing structured constructs, is still rather weak in this area.

Therefore, despite poor text processing facilities, AOS/VS Pascal was the obvious choice of language in which to implement the IV compiler.

5.1. IV Compiler Components

The IV compiler comprises seven integrated modules, each providing a set of related services. Appendix C provides a listing of these services. The complete program listings are found in separate Appendix J. Figure 5.1.1 illustrates the interactions of these modules.



MAIN MODULE
IV SOURCE CODE
LEXICAL ANALYZER
FILE HANDLER
SEMANTIC ANALYZER
STACK HANDLER
DATA DICTIONARY
CODE GENERATOR
AOS/VS FORTRAN 77 & DML CODE
1: Initialize tables of reserved words etc.
2: Open and close all files required by IV
3: Token corresponding to current word
4: Invoke a generator corresponding to token
5: Push and pop expression elements on/off stack
6: Enter data object declarations into dictionary
7: Retrieve information from dictionary on a data object

figure 5.1.1 IV compiler components

[19] Dijkstra, E.W. (1972) *Notes on Structured Programming.*
Wirth, N. (1974) On the Composition of Well-structured Programs. *Comput. Surv.* 6(4)
Referenced by [28]

The main module performs initialization of all tables required by the parser. The values recorded in these tables are the tokens that the lexical analyzer associates with each word that it reads. This module also provides routines for reporting syntactic and semantic errors, and aborting compilation if necessary.

The compiler needs to read and write information from and to various files. The file handler provides routines for opening and closing these files.

The code generator provides basic code generation services - routines for constructing strings of characters, storing them in sequence in a dynamic list structure and finally writing these strings to a file.

A stack handler provides routines for managing a simple LIFO (Last In First Out) stack.

Routines for handling information on data object declarations that are relevant to the compiler are provided by a data dictionary manager.

The lexical analyzer performs lexical analysis on an input string of IV source code. The input string comprises a number of consecutive ordered words, each separated by spaces. The lexical analyzer assigns an unique token to each word as it is read.

The semantic analyzer receives the tokens from the lexical analyzer, checking whether they are in a valid sequence. If the sequence is invalid then an appropriate error message is reported. If the order is valid, then either a call to the relevant code generator is made, or code generation is deferred until some further condition is met, in which case the tokens must be stored on the stack.

5.2. IV Compilation Strategies

The IV compiler employs the LL(1) top-down parsing by recursive descent method. Although not as powerful as some other parsing methods (for example LR parsing) LL parsing is simpler to implement, is more efficient, and allows modifications and extensions to made with relative ease. A parser-generator was not used, although such a tool would have eased implementation effort considerably, due to no suitable system being available on-site, and the semantic complexity of the language.

Since the purpose of this report is not to discuss parsing strategies, the rest of this section is devoted to elaborating on specific features of the implementation.

5.2.1. **The data dictionary** [39,47]

The data dictionary serves as a symbol table handler for the compiler, and provides the compiler with access to the meta data files produced by the DDF and FSED. Unfortunately, the packed meta data (PMD) files, which describe the structure of a database, cannot be accessed directly since information concerning their internal structure is unavailable. See [9,15] for further information.

Instead, the source code listing file (see Appendix F), produced by the DDF [9], is read by the data dictionary which constructs a table internally, recording all the record names, record field names, set names, owner and member record names of each set, set sorting criteria, and the names of sort keys. A similar structure for screen definitions is constructed to record relevant information - field names, field types, field co-ordinates - obtained from the screen definition files.

This information is required by the compiler in order to check that the user has named variables that have been declared, and to determine database navigation through the use of valid set names.

The PMD and screen definition files are used by the DBMS and FSM respectively at run-time for accessing and managing data files and terminal/user interactions.

Figure 5.2.1.1 illustrates the interactions described above.

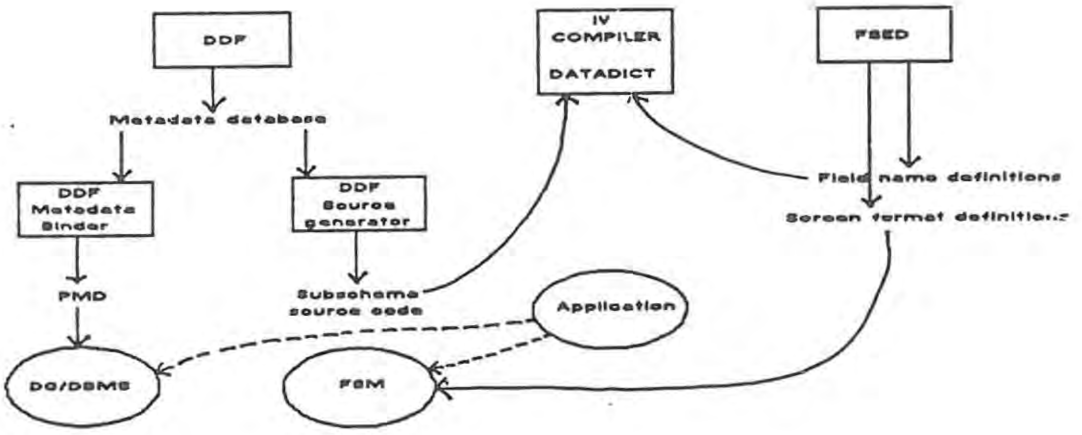


figure 5.2.2.1 IV environment components

5.2.2. Code generation

For most IV statements, FORTRAN code is generated on a token-by-token basis. As a token is received by the syntax/semantic analyzer, an appropriate generator is called which generates the required string of characters. In some cases, however, code generation must be deferred until the semantic analyzer has gleaned further information from the program text. This is particularly relevant when DML statements for record location need to be generated.

Unlike FORTRAN, IV is insensitive to the textual location of IV statements or components of statements. The key word signifying the start of a statement (e.g. IF, DISPLAY) or executable code section (e.g. MODULE), and their corresponding END statements serve as delimiters, separating one statement - or executable code section - from another. Spaces and line feeds serve as separators, which are used to distinguish between identifiers, key words and numeric literals. These features greatly improve program clarity, thus having a beneficial impact on maintenance.

5.2.2.1. Data object declaration.

Declaration of local program variables follow the format of AOS/VS FORTRAN 77 declarations. This results in a simple one-to-one mapping from IV to FORTRAN.

Unlike FORTRAN, IV parameter declarations must specify the parameter's type making it unnecessary to redeclare those parameters. Thus, the following IV text:

```
MODULE 120 ( INTEGER*2 A,B REAL*8 C)
CHARACTER*25 D(10)
    .
    .

END MODULE
```

maps to the following AOS/VS FORTRAN 77 code:

```
SUBROUTINE SXSYS120[20] ( A, B, C )
INTEGER*2 A
INTEGER*2 B
REAL*8 C
CHARACTER*25 D(10)
    .
    .
```

Parameters A and B are redeclared separately rather than as "INTEGER*2 A,B", for ease of implementation.

[20] All identifiers follow strict naming conventions laid down by Steel Information Systems at MS&A. These conventions are listed in the document *Standards for Steel Systems Development* (internal document, MS&A).

The declaration "**SUBSCHEMA** *subschema-name*" maps to the DML statement "**INVOKE** (*subschema-name*)". This non-executable statement copies the subschema source code listing code into the program, thus declaring the UWA records used by the application and the DBMS for data transfers [9,15].

The following FORTRAN declarations are generated for each module that declares at least one screen.

```
REAL*8 RARR(151)
INTEGER*4 FKY,MODE,IPOS,ISTK,IERR,IARR(151)
CHARACTER*50 NAME
CHARACTER*n CARR(151)
```

where n is the field width of the widest character screen I/O field.

These are the parameters used by the screen manager. The number of elements in each of the FSM data buffers (IARR, CARR and RARR) is 151 - which is the largest number that FSED caters for. All real and integer values transferred between the application and FSM require eight and four bytes of storage respectively.

5.2.2.2. Procedural statements

All IV procedural statements can be described in terms of FORTRAN **IF...THEN...ELSE...ENDIF** and **GO TO ...** statements. The following examples illustrate how these procedural statements map into FORTRAN.

1: *IV WHILE statement:*

```
WHILE a_condition DO
      .
      .

END WHILE
```

FORTRAN equivalent:

```
L1:  IF .NOT. a_condition GO TO L2
        .
        .

     GO TO L1
L2:  CONTINUE
```

2: *IV REPEAT statement*

```
REPEAT
  .
  .

UNTIL a_condition
```

FORTRAN equivalent

```
L1:  CONTINUE
     .
     .

     IF .NOT. a_condition GO TO L1
```

3: *IV SELECT statement*

```
SELECT
     WHEN a_condition : statement_clause
     WHEN a_condition : statement_clause
END SELECT
```

FORTRAN equivalent

```
     IF a_condition THEN
          statement_clause
     ELSE
          GO TO L1
     IF a_condition THEN
          statement_clause
     ELSE
          GO TO L1
L1   CONTINUE
```

4: *IV IF statements* map directly into FORTRAN IF statements, each statement conforming to the FORTRAN requirement of one FORTRAN statement per line.

At the start of generating code for an IV procedural construct, the compiler can determine all the unique labels required for that construct. This is achieved by use of a simple counter which is incremented after each label has been generated. Thus, there is no need for backpatching in the case of forward references.

5.2.2.3. Non-procedural statements

An IV non-procedural statement maps down into several FORTRAN statements. Some of these statements perform the command specified by the IV statement, while others, unbeknown to the user, perform basic housekeeping tasks.

Screen management is initiated with the following subroutine call:

```
CALL FFSM(ISTK,FKY,MODE,NAME,IARR,CARR,RARR,IPOS,IERR)
```

where

ISTK	is a record of positions of those fields which have had their contents changed.
FKY	records the value of the function key pressed by the user.
MODE	instructs the screen manager how to perform terminal I/O. Appendix G provides further information on screen modes.
IPOS	informs the screen manager on the field position of the cursor when the screen specified in NAME is first presented.
IERR	is the error code returned by FSM. This parameter will have the value zero if the call to FSM completed successfully.

The IV statement

DISPLAY [EACH] [MENU] Snn (*name_of_screen_field_in_position_x*)
where_clause
: statement_clause
END DISPLAY

will map into the following equivalent FORTRAN code: [17]

```
     IPOS = position_x
     MODE = 2                    { see Appendix G }
L1:  CONTINUE
     where_clause
     CALL FFSM(ISTK,FKY,MODE,NAME,IARR,CARR,RARR,IPOS,IERR)
     IF IERR .GT. 0 GO TO label_of_error_handler
     [ call_logging_subroutine     { see Appendix H }
     check_valid_menu_option       { see Appendix H }
     IF VALID .EQ. "N" GO TO label_of_error_handler ]
                                   { [..] above included only if MENU specified in
                                   header clause }
     MODE = 8                      { see Appendix G }
     statement_clause
     error_handler
     [ GO TO L1 ]                  { included if EACH specified in header }
```

Appendix G illustrates the standard screen layout. Note that the first two screen fields must display the name of the program and the date (see Appendix H) respectively.

Transforming IV transfer functions to their FORTRAN equivalents is rather straightforward, as depicted by the following examples:

STORE *a_record_name*, or
RECONNECT *a_record_name*

map respectively into the following FORTRAN code: [9,15]

```
D    STORE(RECORD=a_record_name,IERR=label_of_error_handler), and
D    RECONNECT(SET=set_name,IERR=label_of_error_handler).
```

The process of translating IV record location qualification expressions into their DML equivalents is beset with problems. Unlike IV and most other modern data manipulation languages, DML does not specify any boolean operators and only one relational operator may be used per qualification expression. DML uses simple relational expressions which are specified as a set of record field names related, via a single relational operator, to a set of corresponding values. If V was the target record type with fields V.1 to V.n, the DML qualification expression has the following format:

> **V.i,...,V.k .EQ.|.NE.|.LE.|.LT.|.GT.|.GE. UWA** where $1 < i,k < n$
>
> The search list **V.i,...,V.k** can be replaced by **SORT KEY**, to indicate that the qualification expression involves only those fields defined (by the DDF) to be keys of a sorted set.
>
> UWA indicates that the values to which the search list is being related to, are stored in the UWA fields corresponding to those named in the search list.
>
> Note that an empty search list can be specified, in which case the position specifiers FIRST, NEXT, LAST, OWNER, PRIOR, or CURRENT will determine which occurrence to locate.

The IV *where clause* also delineates the path to be taken through the database. This is done by specifying a list of qualification expressions, each of which defines the occurrence of the next record type in the network.

The format of the *where clause's* qualification expressions is as follows:

```
WHERE
        FIRST | EACH   record_name1(qualification_expression1)
                                .
                                .
        FIRST | EACH   record_nameK(qualification_expressionK)
                                .
                                .
        FIRST | EACH   record_nameN(qualification_expressionN)
```

The key word FIRST indicates that only a single occurrence is located; EACH indicates a processing loop where all occurrences satisfying the search criteria defined in the qualification expression are retrieved.

The general form of the code generated from an IV data management *where clause* can be expressed as follows: (Appendix D provides a complete example)

```
      FIND FIRST record_name1 satisfying qualification_expression1
      IF NOT found GO TO L2
L1        .
          .
          FIND FIRST record_nameK satisfying qualification_expressionK
          IF NOT found GO TO L3
L4            .
              .
              FIND FIRST record_nameN satisfying qualification_expressionN
              IF NOT found GO TO L6
L5
                  perform transfer function on target record

              FIND NEXT record_nameN satisfying qualification_expressionN
         *    IF NOT found GO TO L6
              GO TO L5
L6            CONTINUE
              .
              .
          FIND NEXT record_nameK satisfying qualification_expressionK+1
     *    IF NOT found GO TO L3
          GO TO L4
L3        CONTINUE
          .
          .
      FIND NEXT record_name1 satisfying qualification_expression1
  *   IF NOT found GO TO L2
      GO TO L1
L2    CONTINUE
```

* Included only if key word EACH is specified in IV qualification expression.

The **IF NOT found GO TO Li** clause is replaced by the END clause of a DML statement. i.e.

FIND(FIRST,RECORD=*record_nameK*,SET=*set_name*,*qualification_expressionK+1*,END=L3)
Navigation in DML requires explicit naming of the sets participating in the search. Since set names are not specified in IV qualification expressions, the compiler must determine which set is to be used from the operands denoted in those expressions. This means that the entire qualification expression must be parsed, each token and identifier name being stored on a stack in reverse order, before the DML statements can be generated.

For example, the qualification expression condition (a>6 OR b<c) AND d=5), is stored as follows:

```
    a   6   >   b   c   <   OR  d   5   =   AND
                                              ↖
                                         Top-of-Stack
```

While constructing the stack, the compiler checks whether one, and only one, of the operands of each relational expression references a database record field. If so, then that operand can be used in the DML search list. A count is kept of the usage of each relational operator. Those record fields participating in relational expressions using the relational operator with the highest usage count will be named in the search list. The search list will be replaced by **SORT KEY** if all the record field operands named in the relational expressions using the "=" operator are defined as keys of a sorted set.

The set will have the record type, whose fields are defined as its keys, as its member, and the record type of interest in the previous qualification expression as its owner. If there is no previous qualification expression then the owner must be the SYSTEM record. This set is then named in the DML statement.

If SORT KEY is not to be used for locating a record then any set can be named provided that that record is defined as the member of the set, and the owner record of that set corresponds to the record named in the previous qualification expression.

Note that in the case of set connection commands (CONNECT, DISCONNECT and RECONNECT), the sort key(s) of the set to which a record is being connected to, or disconnected from, must be explicitly stated in an additional clause to the connection command (see section 4.4.1.2).

The DML search list may not be able to specify all the conditions defined in the IV *where clause* due to the limited expressibility of DML. In those cases where search criteria could not be specified in the DML qualification expression, additional checks would have to be carried out in the host language. Here, the record would have to be retrieved from the database before these checks could be carried out. The following examples illustrate those instances when additional tests are necessary:

Problem: Use of more than one relational operator.

English: locate the first employee whose salary lies between R10000 and R10100.

IV qualification expression:[21]

```
                    .
                    .
    FIRST EMP-REC(SALARY > 10000 AND SALARY < 10100)
                    .
                    .
```

DML and FORTRAN record location:

```
        SALARY = 10000  ! ASSIGN VALUE TO UWA FIELD
    D   FETCH(FIRST,RECORD=EMP-REC,SET=SOME-SET-NAME,SALARY.GT.UWA)
    10  IF SALARY.GT.10100 THEN
    D       FETCH(NEXT,RECORD=EMP-REC,SET=SOME-SET-NAME,SALARY.GT.10000)
            GO TO 10
        ENDIF
```

Problem: Comparison of two fields of the record that is the subject of the search

English: Find the first employee whose pension fund contribution exceeds his U.I.F. contribution.

IV qualification expression:

```
                    .
                    .
    FIRST EMP-REC ( PENSION > UIF )
                    .
                    .
```

[21] Note that this qualification expression will form part of a data management statement

DML and FORTRAN record location:

```
D     FETCH(FIRST,RECORD=EMP-REC,SET=SOME-SET-NAME)
10    IF UIF.GT.PENSION THEN
D           FETCH(NEXT,RECORD=EMP-REC,SET=SOME-SET-NAME)
            GO TO 10
      ENDIF
```

Problem: Use of boolean operators.

English: Find the first employee whose pension fund contribution exceeds R1750 or whose U.I.F. contribution exceeds R2400.

IV qualification expression:

```
                    .
                    .
  FIRST EMP-REC ( PENSION > 1750 OR UIF > 2400 )
                    .
                    .
```

DML and FORTRAN record location:

```
D     FETCH(FIRST,RECORD=EMP-REC,SET=SOME-SET-NAME)
10    IF UIF.GT.1750 .OR. PENSION.GT.2400 THEN
D           FETCH(NEXT,RECORD=EMP-REC,SET=SOME-SET-NAME)
            GO TO 10
      ENDIF
```

Problem: a positional search that must satisfy a condition.

English: Find the manager, whose grade is less than 5, of the first employee whose pension fund contribution exceeds R1750 or whose U.I.F. contribution exceeds R2400. Assume that the MANAGER record is an owner of the EMP-REC record and they participate in the set MANAGER-EMP-SET.

IV qualification expression:

```
                    .
                    .

  EACH EMP-REC ( PENSION > 1750 OR UIF > 2400 )
  OWNER MANAGER ( GRADE < 5 )
                    .
                    .
```

DML and FORTRAN record location:

```
D    FETCH(FIRST,RECORD=EMP-REC,SET=SOME-SET-NAME,END=20)
10   IF UIF.GT.1750 .OR. PENSION.GT.2400 THEN
D        FETCH(NEXT,RECORD=EMP-REC,SET=SOME-SET-NAME,END=20)
         GO TO 10
     ENDIF
15   CONTINUE
D    FETCH(OWNER,RECORD=MANAGER,SET=MANAGER-EMP-SET)
     IF GRADE.LT.5 GO TO 20
     FETCH(NEXT,RECORD=EMP-REC,SET=SOME-SET-NAME,END=20)
     GO TO 15
20   CONTINUE
```

5.2.2.4. Error handling

Calls to the error handler are made only if screen or database management was unsuccessful. In the case of screen management, FSM returns an error code (see Appendix G) in the parameter IERR. Code to test the value of this parameter can be generated immediately after the FSM call has been generated as depicted below:

```
CALL screen_manager
IF IERR .NE. 0 GO TO label_of_error_handler
```

All DML statements have an ERR clause with format [9,15]:

```
ERR = error_label
```

For example,

```
STORE(REC=EMP-REC,ERR=999)
```

If the database manager should signal an error[22], control is immediately passed to *error_label*. The database manager also returns an error code in the system variable DBSTATUS, which can then be checked to determine the course of action.

Appendix E illustrates the code produced for an IV error handler that deals with both screen management and database management errors. Appendix D depicts the case where an error handler has not been specified for a particular non-procedural command and control must then pass to the next error handler.

5.2.2.5. Housekeeping

Several routines (listed in Appendix H) exist for the purposes of basic housekeeping. These routines are usually called from a FORTRAN application before or after specific events occur. Certain housekeeping tasks have been automated, consequently making applications more reliable and consistent.

The following table defines which events require calls to the housekeeping routines to be generated:

EVENT	HOUSEKEEPING
Display a screen or menu	Assign the program name to the first screen field. Assign the current date to the second screen field.
Display a menu	Upon return from FSM, check the current user's privileges on the selected option.
Display a menu, database update	Upon successful completion of operation, call the system logging routine.

22 AOS/VS FORTRAN 77 environment manual

Since housekeeping code is generally included only after a system has been fully tested, these calls are only generated for production systems.

5.3 Related Implementation Issues

Although not included in the specification, portability is an issue to be considered. The database that is supported (DG/DBMS) and the Data Definition (DDL) and Data Manipulation (DML) languages employed by DG/DBMS are based on the 1978 CODASYL specifications. The FORTRAN code produced by the compiler conforms to the full ANSI standard (X3.9-1978), albeit with a few extensions which are well-documented in [14]. The screen generator was developed in-house[23], [16] and [17] providing the interface specifications. IV programs are thus portable (with minor alterations indicated in the system documentation) amongst common systems using CODASYL DBMSs, X3.9-1978 FORTRAN 77 and a stand-alone screen generator.

Currently, no performance evaluation on the compiler, or the code that it produces, has been carried out. Compilation time could, however, be greatly reduced by generating Assembler rather than FORTRAN. It is felt that there is little room for improving application run-time. Most IV procedural statements map directly into equivalent FORTRAN code. IV non-procedural statements are also merely higher level abstractions of blocks of FORTRAN code which occur frequently in existing FORTRAN application programs. Further critical analysis still needs to be performed in view of the performance requirements of the operational environment for which the system was designed.

---

[23] By Steel Information Systems - Middelburg Steel and Alloys

6. CONCLUSION

The rationale for fourth generation environments is simple and easy to explain: skilled computer professionals are becoming increasingly expensive. In order to save on such costs it has become necessary to automate several aspects of programming, making the actual writing of programs a smaller portion of the overall development time of applications.

It has been frequently asserted that analysis work, rather than coding, takes the bulk of the time needed to develop a system (70%-30% [19]), making the claims of huge productivity gains illusory. However, if maintenance effort were also to be considered then the above ratio would change considerably, making analysis relatively less important.

The use of prototyping tools could also be used to reduce analysis time. Such a tool, a graphics language for specifying screen calls, is being considered as an enhancement to the IV environment. The user would be able to see immediately if the system meets his requirements. The IV code generated by the graphics language would also provide the programmer with a skeleton program to which other functions could then be added, thus further reducing overall development time.

Although the work presented in this text represents just the start of an ongoing project, it is felt that the language introduces several elegant programming concepts that offer some improvement over commercial fourth generation languages such as NATURAL and SQL, and certainly considerable advancement over FORTRAN and DML.

Quantitative statistics of IV's impact on programmer efficiency are not yet available, however, favorable reaction has been received from application programmers who have been presented with the proposal. Not only are IV programs considered far more readable than corresponding COBOL or FORTRAN applications, thus having a beneficial effect on maintenance, but it is also felt that the non-procedural statements improve on current methods. The use of declarative facilities will also contribute to system quality through improved reliability attained from the automatic generation of code.

The IV interface to DBMS allows substantially more flexibility than DML in expressing search criteria in the qualification expressions (Appendix D provides a comparative example). McDonald and McNally [31] point out that the usability of a language is proportional to the level of qualification complexity. Since future requirements entail extending the database interface to include relational database managers, a more relational-like view of DBMS will have to be conceived, making the language even more flexible whilst maintaining the run-time performance advantage of the network model.

IV also offers greater symmetry than either NATURAL or embedded SQL. It presents a uniform programming environment rather than the two tiered one found in embedded SQL. It also improves on NATURAL by introducing further restrictions on the scope of variables, forcing the programmer to group logically related operations. For example, those operations that affect data in the database are communicated to the database manager by textually localized code - the *statement clause* of a database management function. The greater freedom afforded the programmer in NATURAL (and other similar languages) is often detrimental to the debugging and maintenance of application programs where data recorded in the database could be affected by assignments occurring in different textual locations in a program, or in several different programs. The IV data dictionary could also provide further help to the programmer by recording the textual location of code that affects database records. Thus, if a particular record indicates incorrect data, the data dictionary provides a convenient debugging facility to trace where in the program text a possible coded error occurred.

In conclusion, this report presents a simple yet effective fourth generation programming language providing the full programming functionality of a procedural third generation language, and supported by a number of tools allowing database manipulation, user interfaces (through terminals), and declarative facilities for describing the objects on which the operations, specified by the language, are performed.

BIBLIOGRAPHY

1. Alberyu, C., Brown, A., et al (1984) A Program Development Tool. *IBM Journal of Research and Development* 28(1) 60-73.

2. Berry, R.E. (1981) *Programming Language Translation.* Ellis Horwood.

3. Bishop,J (1985) Ada - A Hands On Experience. University of the Witwatersrand.

4. Carpenter, M., Hallman, H. (1985) Quality Emphasis at IBM's Software Engineering Institute . *IBM Systems Journal* 24(2) 121-133

5. Chorafas, D.N. (1986) *Fourth and Fifth Generation Languages Vol I.* McGraw-Hill.

6. Christoff, K. (1985) Building a 4GL Environment. *Datamation.* **31(8)** 118-124

7. Cobb, D.H. (1985) In praise of 4GL. *Datamation.* **31(13)** 90-96

8. Cocco, N. & Dulli, S. (1982) A Mechanism for Exception Handling and its Verification Rules. *Computer Languages.* 7(2) 89-101.

9. Data General/DBMS reference manual (1984) Data General Corporation.

10. Data General/SQL User's Manual (1986) Data General Corporation.

11. Duffy, N. (1982) 4GLs : The Quiet Revolution in Information Systems. *Fact & Opinion Paper, Graduate School of Business Administration, University of the Witwatersrand. No. 16.*

12. Fischer, G.E. (1986) Functional Model for Fourth Generation Languages. *Report NBS-SP-500-130 National Bureau of Standards Special Report.*

13. Flores, I. (1981) *Database Architecture.* Van Nostrand Reinhold.

14. Fortran 77 Reference Manual (1981) Data General Corporation.

15. Fortran 77 Environment Manual (AOS/VS) (1984) Data General Corporation.

16. Full Screen Editor. Internal document - Middelburg Steel and Alloys (Pty) Ltd.

17. Full Screen Manager. Internal document - Middelburg Steel and Alloys (Pty) Ltd.

18. Goodenough, J.B. (1975) Exception Handling : Issues and a proposed notation. *CACM.* 18 683-696

19. Grant, F.J. (1985) The Downside of 4GLs. *Datamation.* **31(13)** 99-104.

20. Hammond, L.W. (1982) Management Considerations for an Information Center. *IBM Systems Journal.* **21(2)** 131-162.

21. Henderson, P. (1980) *Functional Programming: Application and Implementation.* Prentice-Hall.

22. IM/DM Fourth Generation Information Management/Data Management (1985) Control Data

23. IMS Application Development Facility II Version 2 (1985). IBM Programming Information.

24. Inmon, W.H. (1986) Fourth Generation Languages : A Management Survey. *Information Strategy Executives' Journal*. 2(4) 4-7

25. Iverson, C.M. (1987) *IV: A Fourth Generation Language*. Proceedings 2nd Conference of M.Sc and Ph.D Research Students.

26. Jones, T.C. (1986) *Programming Productivity*. McGraw-Hill.

27. Kendall, R. (1984) Reusability in Programming - A Survey of the State of the Art. *IEEE Transactions on Software Engineering*. SE-10(5) 488-493

28. Magnenat-Thalmann, N (1982) Choosing an Implementation Language for Automatic Translation. *Computer Languages*. 7(4) 161-170.

29. Martin, J. & McLure, C. (1983) *Software Maintenance*. Englewood Cliffs, Prentice-Hall.

30. Martin, J. (1985) *Fourth Generation Languages Vol I: Principles*. Prentice-Hall.

31. McDonald, N.H. & McNally, J.P. (1982) Query Languages Feature Analysis by Usability. *Computer Languages* 7 103-124.

32. Myers, G.T. (1976) Composite Design Facilities of Six Programming Languages. *IBM Systems Journal*. 15(3) 212-225.

33. Natural (VMS) Programmers Guide (19) Software A.G.

34. Natural Version 2 Planning Guide (1985) Software A.G.

35. Natural Version 2 Concepts & Facilities (1985) Software A.G.

36. Naylor, C. (1983) *Programs that write programs*. Sigma Technical Press.

37. Orman, L. (1983) A Familial Specification Language for Database Application Systems. *Computer Languages* 8(4) 113-124

38. Pazel, C., Malhotra, A., Markowitz, H. (1983) The System Architecture of EASE-E: An Integrated Programming and Database Language. *IBM Systems Journal* 22(3) 188-198

39. Pratt, P.J. & Adamski, J.J. (1987) *Database Systems: Management and Design*. Boyd and Fraser, Boston.

40. Snyders, J. (1984) In Search of a 4GL. *Infosystems*. 31(10) 28-32.

41. Software AG. Information supplied by SPL Ltd. *System Architecture, ADABAS - Advanced Information Management, Predict - The Dictionary, ADABAS - Effective Database Management for the Growing Corporate Environment, Natural - Fourth Generation Application Development System, Natural - Expanding Fourth Generation Technology.*

42. Specter, D. (1985) Language Features to Support Re-usability. *ACM Sigplan Notices* 18(9) 59-61

43. Stay, J.F. (1976) HIPO and Integrated Program Design. *IBM Systems Journal.* 15(2) 143-154

44. Stevens, W. (1982) How Dataflow can Improve Application Development Productivity. *IBM Systems Journal* 21(2) 162-178

45. Stevens, W., Myers, G., Constantine, L. (1974) Structured Design. *IBM Systems Journal.* 13(2)

46. Tennant, R (1987) Data General as a Corporate Solution. Proceedings North American Data General User's Group - Oct 1987 Las Vegas.

47. Uhrowczik, P. (1973) Data Dictionary/Directories. *IBM Systems Journal.* 12(4)

48. Ungar, E. ,Fischer, P. & Slonim,J. (1984) *Advances in Data Base Management Vol II.* John Wiley & sons.

49. Van Leer, P. (1976) Top Down Development Using a Program Design Language. *IBM Systems Journal* 15(2) 155-170

50. Wade, B.W. *et al* (1976) SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control. *IBM Journal of Research and Development.* 20(6) 560-575

51. Winograd, T. (1979) Beyond Programming Languages. *CACM.* 22(7) 391-401

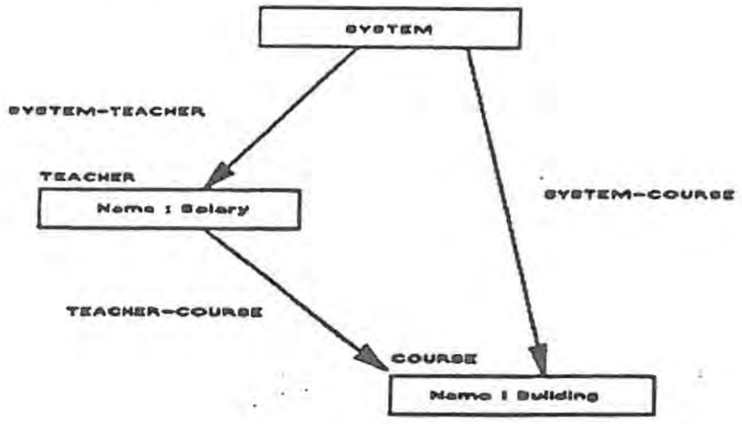
51. Yeh, R. (Ed) (1977) *Current Trends in Programming Methodology Vol I : Software Specification and Design.* Prentice-Hall.

52. Yeh, R. (Ed) (1977) *Current Trends in Programming Methodology Vol II : Data Structuring* Prentice-Hall.

53. Yourdan, E. (1975) *Techniques of Program Structure and Design.* Prentice-Hall.

APPENDIX A

Consider the following logical description of an academic database for the network data model.



Selecting the names of all teachers who teach at least one course in the engineering building, can be accomplished as follows: []

Using DML embedded in COBOL

```
FIND FIRST TEACHER WITHIN SYSTEM-TEACHER.
PERFORM TEACHER-SEARCH
        UNTIL DBSTATUS NOT EQUAL TO ZERO.

TEACHER-SEARCH.
   MOVE SPACES TO BUILDING OF COURSE.
   FIND FIRST COURSE WITHIN TEACHER- COURSE.
   PERFORM COURSE-EVAL
            UNTIL DB-STATUS NOT EQUAL TO ZERO OR
            BUILDING OF COURSE EQUAL TO "ENG".
   IF BUILDING OF COURSE EQUAL TO "ENG"
      GET TEACHER
      DISPLAY NAME OF TEACHER.
   FIND NEXT TEACHER WITHIN SYSTEM-TEACHER.
COURSE-EVAL.
   GET COURSE.
   FIND NEXT COURSE WITHIN TEACHER-COURSE.
```

Using PDML

```
FIND teacher: SYSTEM,SYSTEM-TEACHER,TEACHER-COURSE,
  teacher,course(building = "ENG").
```

Using IV

```
GET TEACHER
WHERE EACH TEACHER
      FIRST COURSE(BUILDING = "ENG")
END GET
```

APPENDIX B

Given the following logical descriptions of a network database and an equivalent relational database:

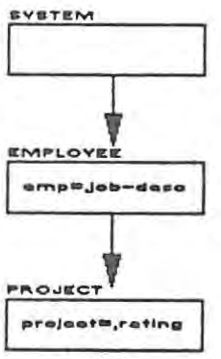


figure A.1 : logical description of network database

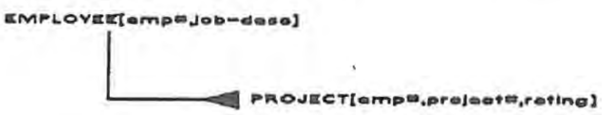


figure A.2 : logical description of relational database

select all employees with a job description code of three whose average project rating exceeds six. The following equivalent solutions are implemented in DML embedded in DG/FORTRAN 77, SQL, IV, and NATURAL Version 1.

Using NATURAL Version 1:

```
FIND EMPLOYEE WITH JOB-DESC = 3
  FIND PROJECT WHERE EMP# = EMP#
    AT END OF DATA MOVE AVER(RATING) TO AV(N2)
  LOOP
  IF AV > 6 DO
     ADD 1000 TO SALARY
     UPDATE SAME RECORD
     END TRANSACTION
  DO END
LOOP
END
```

Using DML embedded in DG/FORTRAN 77:

```
D      INITIATE(ID=IDTX,UPDATE)
D      FETCH(FIRST,REC=EMPLOYEE,SET=SYSTEM-EMPLOYEE,JOB-
       DESC.EQ.3,END=50)
10     SALARY = SALARY + 1000
       TOTAL = 0.0
       COUNT = 0
D      FETCH(FIRST,REC=PROJECT,SET=EMPLOYEE-PROJECT,END=40)
20     TOTAL=TOTAL + RATING
       COUNT = COUNT + 1
D      FETCH(NEXT,REC=PROJECT,SET=EMPLOYEE-PROJECT,END=30)
       GO TO 20
30     IF (TOTAL/COUNT.GT.6) MODIFY(EMPLOYEE)
40     CONTINUE
D      FETCH(NEXT,REC=EMPLOYEE,SET=SYSTEM-EMPLOYEE,JOB-
       DESC.EQ.3,END=50)
       GO TO 10
50     CONTINUE
D      COMMIT
```

Using IV:

```
TRANSACTION

FIND EMPLOYEE
WHERE EACH EMPLOYEE(JOB-DESC=3)
: AVERAGE PROJECT(RATING)
  WHERE EACH PROJECT
  END AVERAGE
  IF AVERAGE>6
     THEN MODIFY EMPLOYEE
          : SALARY = SALARY + 1000
          END MODIFY
  END IF
END FIND

END TRANSACTION
```

Using SQL in a stand-alone mode:

```
UPDATE EMPLOYEE
SET SALARY = SALARY + 1000
WHERE JOB-DESC = 3 AND
        6 < ( SELECT AVG(RATING)
                 FROM PROJECT
                 WHERE EMPLOYEE.EMP = PROJECT.PROJECT#)
```

APPENDIX C

```
Main module : MAIN.MOD
----------------------

        ENTRY PROCEDURE Error ( Error_No : Integer ) ;
        (*
          Report an error message corresponding to the value of Error_N
          Sets a flag to prevent further generation of FORTRAN 77 code.
        *)

        ENTRY PROCEDURE Quit ( Quit_No : Integer ) ;
        (*
          Abort compilation with a message corresponding to the value o
          Quit_No
        *)


File handler : FILER.MOD
------------------------

        ENTRY PROCEDURE Open_File ( File_Type : Integer ) ;
        (*
          Open the file corresponding to the value of File_Type
        *)

        ENTRY PROCEDURE Close_File ( File_Type : Integer ) ;
        (*
          Close the file corresponding to the value of File_Type
        *)


Stack handler : STACK.MOD
-------------------------

        ENTRY PROCEDURE Pop ( VAR An_Element : Pointerto_Stack ) ;
        (*
          Retreive the element at Top Of Stack. Set stack pointer to
          point to 2TOS
        *)

        ENTRY PROCEDURE Push (A_Tag: Tokens ; An_Element: Ident_String
        (*
          Put an element on top of stack. Set stack pointer to point t
          this element
         *)

        ENTRY PROCEDURE Reset_Stack ;
        (*
          Delete all entries from the stack
        *)

        ENTRY PROCEDURE Init_Stack ;
        (*
          Create a dummy stack element that serves as Bottom Of Stack.
        *)


Data dictionary : DATADICT.MOD
------------------------------
```

```
ENTRY PROCEDURE Init_Schema ;
(*
  Set up the database schema structure for the data dictionary
*)

ENTRY PROCEDURE Determine_Set ;
(*
  Determines which set to use for record location on the basis
  of the qualification expression defined in the where clause
  of a data management statement.
  Determines also whether the search should be based on the so
  keys of the determined set, or on search keys, or positional
*)

ENTRY PROCEDURE Init_Find ;
(*
  Set default current root record to "SYSTEM"
*)

ENTRY PROCEDURE Set_Root_Record ( Root_Record : Ident_String )
(*
  Record a new current root record specified by the FIND comma
*)

ENTRY PROCEDURE Unset_Current_Find ;
(*
  Return to the previous current root record. This occurs when
  an END FIND command is encountered
*)

ENTRY PROCEDURE Set_Target_Record ( Target : Ident_String ) ;
(*
  Record the target record specified in the header clause of a
  data management statement
*)

ENTRY PROCEDURE Insert_Program_Var (The_Identifier:Ident_Strin
                  ; Current_Type : Tokens ; Storage : Char )
(*
Record the declaration of parameters and other program variabl
Information recorded is:
    Name of declared variable,
    Type of declared variable, and
    the number of bytes required to store a value of this type
*)

ENTRY PROCEDURE Reset_Declarations ;
(*
  Delete all previous entries in the list of declared program
  variables.
*)

ENTRY FUNCTION Have_Declared ( VAR A_Name : Ident_String ; VAR
                 Of_Type : Tokens ; N_Bytes : Char ) : Boolean
(*
  Used to generate FORTRAN parameter declarations. When parsin
  the IV parameter list, the correct FORTRAN parameter list is
  generated.
  The pameters are inserted in the list which records program
  declarations. Upon completion of the parse of the IV
  parameter list, these entries provide the necessary
  information to declare them
```

```
*)

ENTRY PROCEDURE Init_Program_Variables ;
(*
  Initialise the dynamic list used to record program variable
  declarations
*)

ENTRY PROCEDURE Start_Field_Search ( A_Screen_Name ) ;
(*
  Initialises pointers to the structures containing informatio
  on screen fields - names, types, FSM buffer indexes
*)

ENTRY FUNCTION Retrieve_Field(VAR Field_Name ; VAR F_Type,
                              F_Position : Integer : Boolean
(*
  Returns the information, in the parameters, on the next fiel
  in the list structure containing this information if their i
  such an entry in the structure. If there is no such entry th
  values of the parameters are undefined and the value FALSE i
  returned, else the value TRUE is returned.
*)

ENTRY FUNCTION Corresponds_to(A_UWA_Name, A_Field_Name :
                              Ident_String ) : Boolean ;
(*
  Returns the value TRUE if the value of A_Field_Name
  corresponds to any of the attributes of the record
  specified in the parameter A_UWA_Name ; else FALSE
  is returned
*)

ENTRY PROCEDURE New_Screen ( A_Screen_Name : Ident_String ) ;
(*
  Record the declaration of a screen
  Information generated by FSED is read into a dynamic list
  This information includes:
        The IV screen name,
        the number of named screen names,
        the total number of screen I/O fields defined,
        the name of each field,
        the type of each field variable,
        the position in the FSM buffer arrays for each field
*)

ENTRY PROCEDURE Reset_Screen ;
(*
  delete all entries in the list of declared screens
*)

ENTRY PROCEDURE Init_Screen ;
(*
  Initialise the list used to record screen declarations.
  The list is a simple dynamic list with a dummy head. The mos
  recent declared screen is the first one encountered in the l
*)

ENTRY FUNCTION Check_Id ( Prefix, Variable : Ident_String ;
Expected_Id : Typeof_Ident ; Non_Procedural : Typeof_Block )
: Typeof_Ident ;
(*
  Checks the contextual validity of the variable Prefix.Variab
  Contextual validity is whether the variable conforms to the
```

```
          scope rules of assignmentand reference.
          If the validity check succeeds a check is performed to
          determine whether it has been declared.
          The type of variable - or unknown if checks fail - is return
          Types are:
                Record_Name,
                Record_Field,
                Screen_Name,
                Screen_Field,
                Variable_Name,
                Unknown_Id
        *)

        ENTRY PROCEDURE Set_Current_Screen (Screen_Name: Ident_String)
        (*
          records the screen named in the header clause of a DISPLAY
          statement
        *)

        ENTRY PROCEDURE Screen_Field_Details ( Prefix, Variable :
               Ident_String ; VAR F_Type , F_Position : Integer ) ;
        (*
          retreive details on screen I/O fields
        *)
```

Code generator : CODGEN.MOD

```
        ENTRY PROCEDURE Write_Lines ;
        (*
          write generated FORTRAN 77 code to a file
        *)

        ENTRY PROCEDURE Grabba_Line ;
        (*
          gets a new line onto which to write a FORTRAN 77 statement.
          These lines are kept in a simple FIFO dynamic list. After th
          module has been parsed,
          the code written to these lines is then written to a file on
          secondary storage (Write_Lines)
        *)

        ENTRY PROCEDURE Generate ( A_String : Ident_String ) ;
        (*
          Writes a single word - identifier or numeric literal - to th
          current line in the FIFO line queue
          Ampersands are interpreted as blanks. If the word is followe
          by "^" then a trailing blank is written
        *)

        ENTRY PROCEDURE Generate_String ( A_String, Append_String :
                                                   Line_String ) ;
        (*
          generates a series of words.The end of the string is indicat
          by a "^".
          Append_String is used to generate end of string markers as
          required by AOS/VS
        *)

        ENTRY PROCEDURE Generate_Labl ( The_Labl : Integer ) ;
```

```
(*
  writes a numeric literal in columns 1 to 5 of a new line
*)

ENTRY PROCEDURE Generate_Jump ( To_Labl : Integer ) ;
(*
  writes a numeric literal to the current line
*)

ENTRY PROCEDURE Generate_Continue ;
(*
  generates a FORTRAN CONTINUE statement on a new line
*)

ENTRY PROCEDURE Gen_Cond_Exp ( A_Tag : Tokens ; A_String :
                                              Ident_String ) ;
(*
  generates a conditional expression of a procedural statement
*)

ENTRY PROCEDURE Generate_Mode ( Mode_Number : Integer ) ;
(*
  generates code to set the screen mode to the value of
  Mode_Number
*)

ENTRY PROCEDURE Generate_Screen_Name ( Name : Ident_String ) ;
(*
  generates the FORTRAN 77 application name corresponding to t
  IV name stored in Name
*)

ENTRY PROCEDURE Generate_Ipos ( A_String : Ident_String ) ;
(*
  generates code to set the cursor position in the call to FSM
*)

ENTRY PROCEDURE Generate_FSM_Call ;
(*
  generates the FORTRAN call to FSM
*)

ENTRY FUNCTION Called_Module ( A_String : Ident_String ) :
                                             Ident_String ;
(*
  generates the FORTRAN application module name corresponding
  the value of A_String - the IV module name
*)

ENTRY PROCEDURE Generate_Screen_Field ( F_Type, F_Position :
                                                    Integer )
(*
  Generates the buffer name and index corresponding to an IV
  screen field name
*)

ENTRY PROCEDURE Generate_Record_to_Screen ( A_Screen_Name,
                                A_UWA_Name : Ident_String ) ;
(*
  Given the IV name of a screen in the parameter A_Screen_Name
  code is generated to make assignments
  from the FSM buffers IARR,RARR and CARR to the corresponding
  UWA record fields specified by A_UWA_Name
*)
```

The example below involves updating multiple records subject to a qualification expression involving several different records and attributes.

The compiler will generate the DG/FORTRAN 77 source code with embedded DML commands (indicated by a D in column 7) listed on page D-2

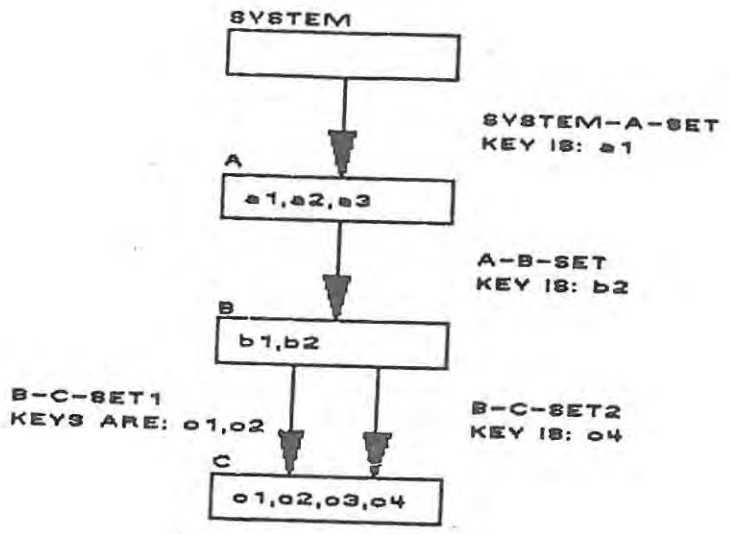


figure D-1 Logical structure of network database

The following IV code section specifies a multiple record location and update operation

```
          .
          .
          .
      MODIFY C
      WHERE EACH  A ( (A1 > 3) AND (A3 < 5.7) )
            FIRST B ( (B1 = 3) OR (B2=3) )
            EACH  C ( (C1 = 4) AND (C2 = B.B1) AND (C3 < 4) )
      :  C2 = S01.NAME
      END MODIFY
          .
          .
          .
```

```
7
          A1 = 3
        D FETCH(FIRST,REC=A,SET=SYSTEM-A-SET,SORT KEY .GT. UWA,
                                  END=2,ERR=99000)
          IF .NOT. (A3 .LT. 5.7) GO TO 2
1         CONTINUE
        D     FETCH(FIRST,REC=B,A-B-SET,END=4,ERR=99000)
              IF .NOT. (B1 .EQ. 3 .OR. B2 .EQ. 3) GO TO 4
3                 C1 = 4
                  C2 = B1
        D         FETCH(FIRST,REC=C,SET=B-C-SET1,
                             SORT KEY .EQ. UWA,END=6,ERR=99000)
                  IF .NOT. (C3 .LT. 4) GO TO 6
5                            C2 = CARR(1) ! ASSUME NAME=CARR(1)
        D                    MODIFY(REC=C,ERR=99000)
                  C1 = 4
                  C2 = B1
        D         FETCH(NEXT,REC=C,SET=B-C-SET1,
                             SORT KEY .EQ. UWA,END=6,ERR=99000)
                  IF .NOT. (C3 .LT. 4) GO TO 6
                  GO TO 5
6             GO TO 3
          A1 = 3
        D FETCH(NEXT,REC=A,SET=SYSTEM-A-SET,SORT KEY .GT. UWA,
                                  END=2,ERR=99000)
          IF .NOT. (A3 .LT. 5.7) GO TO 2
          GO TO 1
2         GO TO 7
99000     GO TO 99001   ! EXIT TO NEXT ERROR HANDLER LOCATED 99001
7         CONTINUE
```

4: IV source code listing in file IVCRQ000P

```
    MODULE 010
       .
       .
       .

      DISPLAY EACH S01(CRQ01-REQNO)
      WHERE CRQ01-CC = "XXX"
      :   IF CRQ01-CC = "XXX"
             THEN EXIT
          END IF
          MODIFY CRQ01-CHANGE-REC
          WHERE FIRST(CRQ01-REQNO = S01.CRQ01-REQNO)
          :   CRQ01-CHANGE-REC = S01
          END MODIFY

          WHEN IERR = 4 : MESSAGE = "Invalid Format file"
                          EXIT
          WHEN DBSTATUS = 17202 : MESSAGE = "Reenter data-Update
                                                        Conflict"
      END DISPLAY
       .
       .
       .

    END MODULE
```

The following FORTRAN 77 code will be generated from the above:

```
      SUBROUTINE SXCRQ010
          .
          .
      IPOS = 3
      NAME = "CRQ010S01"
      IMODE = 2
1     CONTINUE
      CARR(3) = "XXX"
      CALL FFSM(ISTK,FKEY,IMODE,NAME,IARR,CARR,RARR,IPOS,IERR)
      IMODE = 8
      IF IERR <> 0 GO TO 2
      IF CARR(3) = "XXX" THEN
         GO TO 3
      ENDIF
      CRQ01-REQNO = IARR(1)
      FIND(FIRST,REC=CRQ01-CHANGE-REC,SET=CRQ-SYS-01CHANGE-RQSET,
     * END=4,ERR=2)
      CRQ01-CC=CARR(3)
      CRQ01-NAME=CARR(4)
      CRQ01-REQNO=IARR(1)
      CRQ01-SYSNAM=CARR(5)
      CRQ01-DESC=CARR(6)
      CRQ01-STATUS=CARR(7)
      CRQ01-TARGET-DATE=CARR(8)
      CRQ01-RESP-SUP=CARR(9)
      CRQ01-RESP-PROG=CARR(10)
      CRQ01-REJECT-DESC=CARR(11)
      MODIFY(CRQ01-CHANGE-REC)
2     IF IERR = 4 THEN
         MSSG = "Invalid format file"
         CALL FSMMSG(NAME)
      ENDIF
      IF DBSTATUS = 17202 THEN
```

```
            MSSG = "Reenter data-Update Conflict"
            CALL FSMMSG(NAME)
         ENDIF
         GO TO 1
3        CONTINUE
            .
            .
         END
```

Naming conventions

All application systems follow strict standard naming conventions, laid down by Steel Information Systems, to aid reliable system documentation. These conventions provide a set of simple transformation rules for automatically translating IV identifiers into their equivalent FORTRAN counterparts.

IV modules are kept in files with the following name format:

IVsysnnnP	for production systems
IVsysnnnT	for test systems
IVsysnnnN	for new development

where
sys is the system name
nnn is a unique three digit number assigned to each file

All IV modules within a single file must be assigned unique three digit names - mmm. These module names map down into the following equivalent FORTRAN subroutine names: SXsysmmm. For example, **MODULE 010** stored in file IVMCS000P will map into **SUBROUTINE SXMCS010**, which will be stored in the file SXMCS010.F77, and its object file in the production directory.

Display record names have the following format: Stt, where tt is a unique two digit name assigned to the display record. The equivalent FORTRAN name is sysmmmStt. Thus display record S03 declared in the above module has the equivalent FORTRAN name MCS010S03.

```
                DB.DDF
          SCHEMA SOURCE LISTING

SCHEMA NAME IS ":DISK5:DBT:SXABC:SXABC_DBT"

***********************************************

RECORD NAME IS X
  02 X1  PICTURE IS X(3)
  02 X2  PICTURE IS 99

-------------------------------------------------

RECORD NAME IS Y
  02 Y1 PICTURE IS 99
  02 Y2 PICTURE IS X(4)
  02 Y3 PICTURE IS 9

------------------------------------------------
************************************************

SET NAME IS ABC-SYS-X-SET
  OWNER IS SYSTEM
  MEMBER IS X
    INSERTION IS AUTOMATIC
    RETENTION IS MANDATORY
    MEMBER LIMIT IS NONE
    ORDER IS SORTED BY KEY ASCENDING
       SORT KEY IS : X2
       DUPLICATES ALLOWED

------------------------------------------------

SET NAME IS ABC-X-Y-SET
OWNER IS X
MEMBER IS Y
  INSERTION IS AUTOMATIC
  RETENTION IS MANDATORY
  MEMBER LIMIT IS NONE
  ORDER IS SORTED BY KEY ASCENDING
     SORT KEYS ARE :
         Y1
         Y3
     DUPLICATES NOT ALLOWED

-------------------------------------------

         ** END OF SCHEMA SOURCE LISTING **
```

APPENDIX G

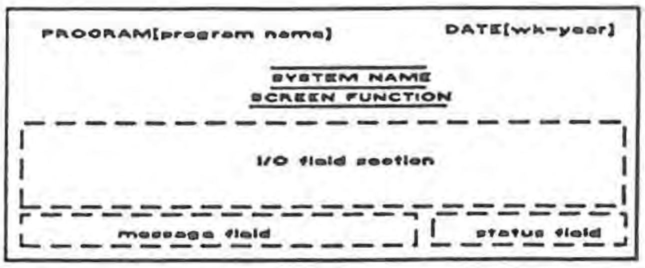


figure G.1 Standard screen layout

1: MODE

VALUE	DESCRIPTION
1(7)	Generate a new screen, all alphanumeric field are set to spaces and numeric fields set to zeros The cursor is placed at the first input field an field control is enforced.
2(8)	Generate a screen. The values in all arrays ar displayed, the cursor is placed according to th value of IPOS. Field control is enforced.
3(9)	As for MODE=2,except field control is not enforce
4	Writes the value of NAME to status field if IPO is zero, otherwise a standard system message i written.
5	Writes the value of NAME to the message field
6	Initialize the status and message fields.

The values in brackets operate in the same way as the corresponding modes except that the screen is not redrawn.

2: IPOS

For MODES 2,3,8 and 9, this should contain the field number where the cursor is to be positioned when the screen is displayed.
For MODES 1,2,6 and 7 it should contain zero.
For MODE 4 it should contain zero to display the value of NAME in the status field, or 1-6 for the following standard system messages:

1	SYSTEM READY
2	*PLEASE WAIT*
3	*COMPUTING*
4	*ACCESSING DATABASE*
5	*PRINTING*
6	*SORTING*

3: NAME

For MODES 1,2,3,7,8 and 9, it should contain the name of the screen to be displayed.
For MODE 4 the value is displayed in the status field.
For MODES 5 and 6 the value is displayed in the message field

4: IERR

0	No errors
1	MODE not in range 1 to 9
2	Message in NAME too long
3	IPOS not in range 0 to 6 for MODE 4
4	Screen definition file specified in NAME invalid
5	Help was requested
6	Format file in NAME was not created by FSED
7	IPOS out of range for MODES 2,3,8,9

APPENDIX H

Calls to these subroutines are automatically generated by the IV compiler in order to perform housekeeping tasks.

MCPLOG:

Creates a record in the system log file

CALL MCPLOG(SYS,MODNO,DESC,MESS,TYPE)

SYS	system name, e.g. MCS
MODNO	module number, e.g. 010
DESC	summary description, generally IV key word, e.g. "DISPLAY","MODIFY" etc
MESS	message description,typically the name of record being manipulated - MCS010S01,MCS01-PRINT-REC. Alternatively any comment immediately following the record name in a header clause will be used for the message description.
TYPE	second letter of system name used for identification purposes.

GETUSAD:

Obtains the ID code of the user of the system

CALL GETUSAD(PGM,USER,ADID,IERR)

PGM	program name
USER	user name
ADID	user identification code
IERR	error code

CHECKMEN:

Checks user privileges on a selected menu option.

CALL CHECKMEN(ADID,SCRN,IARR,VALID)

ADID	user identification code obtained from call to GETUSAD
SCRN	screen name
IARR	selected menu option
VALID	"Y" is a valid option, "N" is not a valid option

WEEKYEAR:

Gets the current week and year to be displayed in the second field of each screen

CALL WEEKYEAR(CARR)

CARR - week/year

APPENDIX I

The following is an abbreviated list of the syntactic forms of the language expressed in EBNF notation.

```
<PROGRAM> ::=
          <MODULE> { <MODULE> }
<MODULE>  ::=
          MODULE <ID> [ ( <DECLARATION_LIST> ) ]
          [ <DECLARATION_LIST> ]
          [ <STATEMENT_SEQUENCE> ]
          END MODULE
<DECLARATION_LIST> ::=
          <DECLARATION> { <DECLARATION> }
<DECLARATION> ::=
          <TYPE_ID> <NAMED_ID> { , <NAMED_ID> }
<NAMED_ID> ::=
          <ID> [ ( <INTEGER> ) ]
<STATEMENT_SEQUENCE> ::=
         <TRANSACTIONS> | <NON_DATA_MANAGEMENT_STATEMENTS>
         { <TRANSACTIONS> | <NON_DATA_MANAGEMENT_STATEMENTS> }
<TRANSACTIONS> ::=
         TRANSACTION <STATEMENT_LIST> END TRANSACTION
<STATEMENT_LIST>
         <NON_DATA_MANAGEMENT_STATEMENT> | <DATA_MANAGEMENT_STATEMENT
         { <NON_DATA_MANAGEMENT_STATEMENT> | <DATA_MANAGEMENT_STATEMENT }
<UPDATE_RETREIVE_STATEMENT_LIST> ::=
         <NON_DATA_MANAGEMENT_STATEMENT> { <NON_DATA_MANAGEMENT_STATEME
<NON_DATA_MANAGEMENT_STATEMENT> ::=
         <ASSIGNMENT> | <SCREEN_MANAGEMENT> | <PROCEDURAL>
<ASSIGNMENT> ::=
         <ID> = <EXPRESSION>
<SCREEN_MANAGEMENT> ::=
         DISPLAY [ EACH ] <ID> [ ( <ID> ) ] [ WHERE <STATEMENT_SEQUENCE>
<RESTOF_SCREEN> ]
<RESTOF_SCREEN> ::=
         END DISPLAY | : <STATEMENT_SEQUENCE> END DISPLAY
<PROCEDURAL> ::=
         <IF_STATEMENT>   |   <SELECT_STATEMENT>   |   <WHILE_STATEMENT>   |
<UNTIL_STATEMENT>
<IF_STATEMENT>* ::=
         IF <CONDITION> THEN <STATEMENT_SEQUENCE>
         [ ELSE STATEMENT_SEQUENCE ] END IF
<SELECT_STATEMENT>* ::=
         SELECT <WHEN_CLAUSE> [ ELSE <STATEMENT_SEQUENCE> ] END SELECT
<WHILE_STATEMENT>* ::=
         WHILE <CONDITION> DO <STATEMENT_SEQUENCE> END WHILE
<UNTIL_STATEMENT>* ::=
         REPEAT <STATEMENT_SEQUENCE> UNTIL <CONDITION>
<DATA_MANAGEMENT>  ::=
         <UPDATE_RETREIVE> | <CONNECTION> | <LOCATION> | <DELETION> | <UTILIT
<UPDATE_RETREIVE> ::=
         <TRANSFER_FUNCTION> <ID> <WHERE_CLAUSE> [ <RESTOF_UPDATE_RETREIVE
```

```
<RESTOF_UPDATE_RETREIVE> ::=
        END <TRANSFER_FUNCTION> |
        : <UPDATE_RETREIVE_STATEMENT_LIST> END <TRANSFER_FUNCTION>
<CONNECTION> ::=
        ( RECONNECT <ID> <WHERE_CLAUSE> <FROM_CLAUSE> <TO_CLAUSE> ) |
        ( DISCONNECT <ID> <WHERE_CLAUSE> <FROM_CLAUSE> ) |
        ( CONNECT <ID> <WHERE_CLAUSE> <TO_CLAUSE> )
        USE KEY <ID>
<LOCATION> ::=
        FIND <ID> <WHERE_CLAUSE> : <STATEMENT_SEQUENCE> END FIND
<DELETION> ::=
        ERASE <ID> <WHERE_CLAUSE> END ERASE
<UTILITY> ::=
        <UTILITY_FUNCTION> <ID> ( <ID> ) <WHERE_CLAUSE> END <UTILITY_FUNCTI
<FROM_CLAUSE> ::=
        FROM <ID> <WHERE_CLAUSE>
<TO_CLAUSE> ::=
        TO <ID> <WHERE_CLAUSE>
<WHERE_CLAUSE> ::=
        WHERE <WHERE_SEQUENCE>
<WHERE_SEQUENCE> ::=
        <WHERE_STATEMENT> { <WHERE_STATEMENT> }
<WHERE_STATEMENT> ::=
        <SELECTOR> [ <RESTOF_WHERE> ]
<RESTOF_WHERE> ::=
        <ID> ( <CONDITION_EXPRESSION> )
<WHEN_CLAUSE> ::=
        <WHEN> { <WHEN> }
<WHEN> ::=
        WHEN <CONDITION> : <STATEMENT_SEQUENCE>
<TRANSFER_FUNCTION> ::=
        GET | MODIFY | STORE
<UTILITY_FUNCTION> ::=
        AVERAGE | COUNT | MAX | MIN | TOTAL
<SELECTOR> ::=
        EACH | FIRST | OWNER
```

* If a sentence generated by these productions is derived from the <TRANSACTIONS> production (i.e. a procedural statement is textually located within a transaction statement) then the <STATEMENT_SEQUENCE> non-terminal is restricted to generating <NON_DATA_MANAGEMENT_STATEMENTS> non-terminals.

APPENDIX J

This appendix is provided under a seperate cover. It presents source code listings of the seven IV compiler modules, the screen formatter (FSED, and the screen manager (FSM). This appendix is available from:

The Department of Computer Science
Rhodes University
P.O.Box 94
Grahamstown
Republic of South Africa