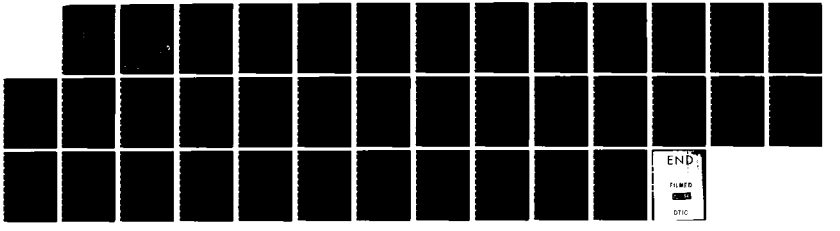
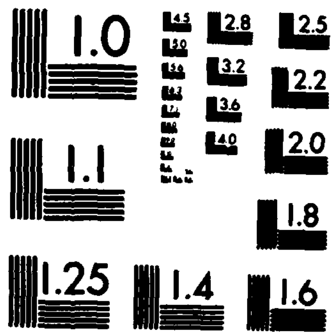
AD-A137 159 AN ANALYSIS OF APPLICATION GENERATORS(U) UNIVERSITY OF 1/1
SOUTHERN CALIFORNIA LOS ANGELES DEPT OF COMPUTER
SCIENCE E HOROWITZ ET AL. MAR 83 TR-83-208
UNCLASSIFIED AFOSR-TR-83-1310 AFOSR-82-0232 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AN ANALYSIS OF APPLICATION GENERATORS

by

Ellis Horowitz
Alfons Kemper
Balaji Narasimhan

TR-83-208

March 1983

COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF SOUTHERN CALIFORNIA
LOS ANGELES, CALIFORNIA 90089-0782

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): TR-83-208	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 83-1310
6a. NAME OF PERFORMING ORGANIZATION: University of Southern California; 6b. OFFICE SYMBOL (If applicable):	7a. NAME OF MONITORING ORGANIZATION: Air Force Office of Scientific Research
6c. ADDRESS (City, State and ZIP Code): Computer Science Department University Park, Los Angeles CA 90089-0782	7b. ADDRESS (City, State and ZIP Code): Directorate of Mathematical & Information Sciences, Bolling AFB DC 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR; 8b. OFFICE SYMBOL (If applicable): NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: AFOSR-82-0232
8c. ADDRESS (City, State and ZIP Code): Bolling AFB DC 20332	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
61102F	2304	A2	

11. TITLE (Include Security Classification): AN ANALYSIS OF APPLICATION GENERATORS

12. PERSONAL AUTHOR(S): Ellis Horowitz, Alfons Kemper, Balaji Narasimhan

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	MARCH 83	35

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	

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

The continuing development of higher order programming languages has not yielded major productivity improvements in the software development process. One often mentioned mechanism for achieving significant orders of improvement are application generators, such as RAMIS, NOMAD, and FOCUS. These systems have been applied to data intensive business applications with phenomenal success. The purpose of this paper is to present the basic components of application generators and show why they yield such large productivity increases in the edp environment. The authors investigate the meaning of nonprocedural programming and show how it exists in current application generators. Then they analyze the possibility of extending application generators so that they may be used for non-edp type applications.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☒ DTIC USERS ☐	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Dr. Robert N. Buchal	(202) 767- 4939	NM

an analysis of application generators

Ellis Horowitz, Alfons Kemper, and Balaji Narasimhan

Computer Science Department

University of Southern California

Los Angeles, California 90089

This work has been supported by the Air Force Office of Scientific Research under Grant no. AFOSR-82-0232

Abstract: The continued development of higher order programming languages has not yielded major productivity improvements in the software development process. One often mentioned mechanism for achieving significant orders of improvement are *application generators*, such as RAMIS, NOMAD, and FOCUS. These systems have been applied to data intensive business applications with phenomenal success. The purpose of this paper is to present the basic components of application generators and show why they yield such large productivity increases in the edp environment. We investigate the meaning of nonprocedural programming and show how it exists in current application generators. Then we analyze the possibility of extending application generators so that they may be used for non-edp type applications.

Table of Contents

1. Motivation and Focus

The past quarter century has witnessed amazing improvement in our ability to fabricate sophisticated hardware devices. The growth in the number of active elements per chip has been doubling every few years for the past twenty five years. But this exponential growth has in no measure been matched by productivity improvements at the software level [1]. One result of this fact is that the cost of a computer system has shifted from being almost entirely hardware related to being almost entirely software related. The distribution of software cost versus hardware cost of a typical system was given by Boehm [2] and is shown in the diagram of Figure 1.1.

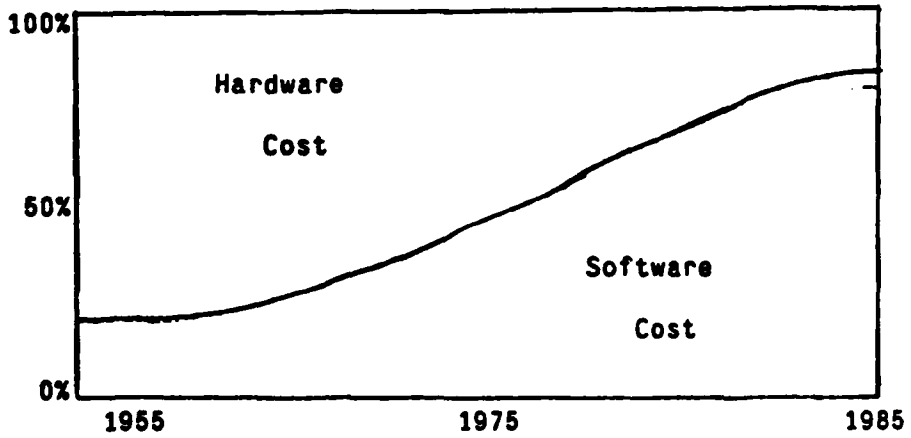


Figure 1.1: Distribution of Software Versus Hardware Cost
For a Typical Computer System

As pointed out by McCracken [3], todays software developers are facing the following challenges:

1. Increased demand for new applications
2. Increased complexity of new applications
3. Increased cost and decreased availability of skilled people

It is extremely unlikely that these demands can entirely be met by the careful application of conventional software engineering methodologies and tools. What we need are techniques that gain us *orders of magnitude* improvements in productivity. A recent IEEE meeting on software productivity [1] said it this way:

> Every panel observed that major increases in productivity for software engineering will come about only through elimination of the number of skilled man hours required to

produce new software. This reduction can occur by: a) reducing the cost of each step in the development/maintenance life cycle, b)eliminating need for a step in the life cycle, c) reducing the number of iterations back through life cycle steps, or d) reducing the cost of impact of change.

One approach to meet these challenges, which we investigate in this paper, are application generators. Application generators, sometimes abbreviated AG, are software systems geared primarily to support data intensive application development. AG's provide a very high level, special purpose "programming" language with a user friendly syntax. Therefore AG's can easily be applied by the end-user, who is usually not sophisticated in computer skills.

Conventional wisdom holds that the software development process can be viewed as a series of steps, consisting of: system specification, architectural design, detailed design, coding, testing and debugging, documentation, and maintenance. When one is using an application generator he begins by specifying a very limited prototype of the desired application and then incrementally extends and modifies the prototype until it meets all of the requirements. This methodology has several implications on the conventional life-cycle process. One, there is no coding phase in the usual sense. The software specification is literally turned into an executable program. Two, testing and maintenance are significantly reduced because the programmer returns to the specification when changes have to be made. This specification is expressed as an application generator program, and as such it is at a level much higher than a program written in a conventional programming language. Therefore the testing and maintenance phases are simplified. Three, documentation is aided by the fact that the program is easily readable. Four, the high level nature of the language permits less sophisticated end-users to program directly. For all of these reasons people have observed that application generators, where appropriate, offer large degrees of improvement in software development time [4].

But can the success of application generators in the world of edp be brought to other environments? This sentiment was suggested recently by the DoD Software Initiative [5] which cited application generators as one means for improved DoD software productivity. In this paper we intend

to look closely at some of the existing application generators to see precisely what facilities they offer and in what form. In Section 2 we will describe the basic components that most of the existing systems share. Then in Section 3 we hypothesize a generic application generator. Using this generic system we discuss the language features it offers and write several programs as demonstrations. Then in Section 4 we contrast application generators to general purpose programming languages and discuss what would have to be done to create a hybrid of these two software systems.

2. Todays Application Generators

Application Generators have their origin in the early report generator systems, such as IBM's RPG [6]. These software packages let the user generate reports at a very high level of description, permitting him to avoid having to deal with low level issues such as page layout and data representation. Todays application generators are substantial extensions of these report generators. They address the need of the business world to have a uniform approach to data intensive applications, i.e. database management, data manipulation, and data retrieval in the form of simple database queries as well as the generation of sophisticated reports. Contemporary application generators may also make use of the more recent hardware developments facilitating graphics support. In addition some AG's include software modules for very specialized applications, such as financial modelling or statistical analysis packages. In summary application generators typically consist of the following modules:

1. database management system
2. report generator
3. database query language
4. graphics package
5. special purpose software

2.1. Examples of Commercially Available Application Generators

In order to get a better idea of the current state of application generators, we investigated the following systems: NOMAD, RAMIS, FOCUS, ADF and dBASE II. In this section we briefly summarize their attributes.

NOMAD:

NOMAD [3] was developed by National CSS, starting in 1973. Recently NOMAD has been extended and is now available as NOMAD2 [7]. NOMAD2 supports three kinds of database structures:

- relational
- hierarchical
- hybrid combinations of hierarchical and relational structure

The hybrid database structure incorporates both hierarchical as well as relational features. This was done in order to combine the efficiency of the hierarchical model with the flexibility of data access of the relational system. In order to provide efficient and flexible data retrieval for a variety of applications NOMAD supports a wide range of access methods such as keyed access, balanced tree indexing, table look ups, etc. Thus NOMAD has the full capabilities of a database management system, including data manipulation, data integrity, and data security features.

In NOMAD the end-user can generate reports from the database using the list command. The underlying data records of the report can be sorted, screened, and totalled by applying some very high level language constructs such as the "select...where..." command for data screening NCSS has added high level graphics features to NOMAD2 to generate graphics reports on the screen. Most of the scaling is done automatically by the system, but, if the user wishes, he can choose his own scaling factors, thereby overwriting the predefined system parameters. Also the user specified headings and legends are automatically put into place by the system.

FOCUS:

FOCUS [8], developed by Information Builders,Inc., is one of the most extensive application generators available. It is based on a hierarchical database management system and provides an

interactive data entry language, called FIDEL. Besides the high level report generator and a sophisticated terminal graphics package it has built in a wide range of special purpose software for business applications. This covers statistical analysis functions and a financial modelling language.

For the more experienced user FOCUS has a host language interface to facilitate the access to FOCUS files from a general purpose programming language, like FORTRAN or COBOL. Also the user can redefine the syntax of FOCUS to suit his needs.

RAMIS:

RAMIS [9] was developed by Mathematica Products Group,Inc. RAMIS is designed specifically for IBM mainframe computers. The RAMIS database is structured as a network of data segments, where the segments can be in different files. In addition RAMIS supports a purely hierarchical data model. Besides its own files RAMIS can also interface to DL1 and IMS files. The Records Management System provides the user with a nonprocedural language for retrieving and processing data records. The Report Preparation System of RAMIS is designed in the form of an English-like language and lets the user retrieve, sort, calculate, and format data into tabular or graphic reports. In RAMIS the report generator and the graphics language were designed to have a unified syntax.

dBASE II:

In recent years several systems for application generation have been designed for personal computers [10]. One of these systems is dBASE II [11], which is a relational database system for microcomputers. It was developed by Ashton-Tate in 1980 and runs under CP/M. In addition to the database operations such as addition, deletion, update of records that one can perform using the data manipulation language of dBASE II, one can also generate reports from one or more databases. The report generation facility is very concise and consists of a set of prompts from the system to the user, about the different aspects of the report such as report heading, column headings, totals on columns etc. The command language of dBASE II has the syntax of an Algol-like language. dBASE II also provides a full-screen editor to set up a screen format for use as a data entry facility. Note that this entire system costs significantly less than $1000.

Application Development Facility (ADF):

Application Development Facility [12] was developed by IBM as an "installed user program" to be used with their IMS database management system. Applications developed using ADF have a common overall structure. Such an application is produced as a set of program modules tailored by the applications developer to suit his specific application. The modules contain the programming logic for the following tasks.

- dialog management
- data access
- application logic
- control of the interaction of the above modules

A Transaction Driver directs the execution of the application program so that the menus, data displays, and messages are joined to produce a complete application. ADF also supports non-conversational and batch mode applications.

An application developer tailors the modules to suit his application by supplying a set of rules using a simple, English-like language as input to a component of ADF, called the Rule Generator. These rules are used internally by the transaction driver to control the functions of the modules at execution time. Where programming is required by the special nature of an application, such programming can be interfaced with the above modules.

2.2. The Basic Components of Application Generators

In this subsection we will inveatigate in detail the basic components of an application generator which are present in most of the commercial systems.

2.2.1. The Database Management Component

All commercially available AG's have their own DBMS, and in addition support access to external files as well. Typically the underlying data model is either hierarchical or relational. A database usually consists of two files:

- the master file and

- the data file

The master file contains user entered data format information describing a particular database, i.e. type definition of fields, field names, relationship among different segments, etc. Creation of this master file is usually done interactively, such that the system displays some skeleton form for the master file and the user, in this case the database administrator, has to fill in the blanks.

The data file contains the actual data, which is entered and modified via the database query language(see section 2.5). In order to make access to this data as efficient as possible the user can specify in the corresponding master file what access method is to be used and he can also specify the key fields of the database, if any.

2.2.2. The Report Generator Component

One of the essential features of all application generators is the report generator feature. These facilities are provided in the form of a special purpose sublanguage. The language of the report generator is often characterized as *non-procedural*. This, however, does not mean that there are no subroutines; in section 3 we will show an example of the use of a subroutine in report generation. The characterization of the report generator as non-procedural is used in the sense of *very high level*. This means that the programmer has to specify only the major steps of the computation but is not concerned about the low level details, such as data representation or the exact sequence of computation. In the literature, notably Leavenworth and Sammet [13] and Leavenworth [14], non-procedural languages are often characterized by the fact that the user has to specify *what* he wants to be done. But he can leave out the details of *how* the system is to accomplish this task. For this reason the term *goal oriented* language might be more appropriate to define the nature of these non-procedural languages. Sammet [13] pointed out that all these terms are relative and depend on the state of the art. For example, the first high level programming languages like Fortran and Algol60 were considered as very high level with respect to assembly languages, which again were viewed as high level relative to machine languages.

a)

```
file EMPLOYEES
list NAME SALARY
```

b)

```
<open file EMPLOYEES>
for i:=1  to MaxNumOfEmpl  do
     begin

     <retrieve i^th data record using
       some existing retrieval path>
      <extract the NAME and SALARY fields>
       writeln(NAME[i],SALARY[i])

     end
<close file EMPLOYEES>
```

Figure 2.1: Program Fragments in a) Report Generator and b) Conventional Programming Language

To get a better understanding of the difference between using a non-procedural language, and a conventional general purpose programming language, let us look at the program fragments of Figure 2.1. Both programs accomplish the same task, i.e. generating a list of all employees and their respective salary from the database EMPLOYEES. Fig. 2.1(a) shows the program as it would appear in an Application Generator whereas the much longer version, shown in Fig. 2.1(b), outlines the basic steps of the program written in some conventional Pascal-like programming language. Note that the program in Fig. 2.1(a) is complete whereas Fig. 2.1(b) only shows an outline of the program to be written. Two major low level details which are removed from the programmer's concern when he uses an application generator are: the specification of explicit iteration over all data records and the specification of the retrieval paths to obtain the data fields (e.g. NAME and SALARY) from the data records. In a report generator the user can rely on *associative referencing*. This is a term describing the accessing of data based on certain characteristics associated to it, like field name or range of possible values. This is in contrast to explicitly traversing some existing retrieval path or performing a search over all possible elements in a certain data set, like a database relation. The user of an application generator is not concerned with how the system is actually managing to retrieve the data. Furthermore the details of the report layout and conversion of the data to the appropriate output format are left to the system. In summary we can say that in a conventional programming language one has to specify the program more in terms of how the system is supposed to do the computation.

In an AG the user writes more in terms of what the outcome of the program should look like.

The syntax of the report generator sublanguage is typically very much like natural (English) language. The reason for this is twofold; for one the report generator is mostly used in business applications where Cobol was, and still is, the primary programming language. It was felt that the English-like syntax would ease the transition from Cobol to an application generator for most programmers in that application domain. Another reason is that application generators were designed to be used by the end-user, i.e. a business person with none or very little data processing experience. The English-like syntax might help these users to adapt to the new system more quickly.

2.2.3. The Graphics Package Component

Recently application generators have begun to include a sophisticated graphics package which, just like the report generator, interfaces to the DBMS. The graphics package is actually a special form of the report generator, with the distinction that a *graphical report* is produced rather than tabular output. For ease of use the syntax of the graphics language is basically the same as for the report generator. The main difference is that the user has to specify in what kind of a graph he wants the information to be reported. Typically application generators provide the following five kinds of graphs:

- connected point graphs
- histograms
- barcharts
- scatter diagrams
- piecharts

With the growing development of graphics hardware and software it can be foreseen that the graphics software in an application generator will become more sophisticated. One important development on the horizon is automatic scaling and headings for graphs. Also color graphics and 3 dimensional representation of the information is already possible. Optimally an application generator is connected to a graphics display terminal as well as a plotter to facilitate interactive output on the

screen and also hardcopy output. For further explanation of the graphics package with an extensive set of examples we refer the reader to section 3.

2.2.4. *The Database Manipulation Language Component*

Ideally the database manipulation language should use the same syntactical structure as the other modules of the AG, namely the report generator and the graphics package. The database manipulation language must provide for interactive as well as batch processing of the database modifications to be done. It has to include functions for the following operations:

- inserting data
- deleting data records
- updating data
- retrieving and listing of data records
- statement of consistency constraints
- statement of authority constraints

If the user wants to interactively edit the database the easiest syntactical way seems to be one in which the system prompts the user for the necessary information which needs to be specified for the corresponding operation. This can be done by displaying a skeleton table of the corresponding database and letting the user fill in the blanks or modify the existing entries. This approach was taken in the design of Query-By-Example, which was developed at IBM [15]. The data manipulation language is mostly menu driven, i.e. the possible operations supplied by the system are being selected via a menu. This eliminates the need for the user to memorize the complete syntax of the system.

In addition to editing the data the user has also the possibility to modify the definition of the database. In this case he has to edit the master file, i.e. the file with the type definition of the corresponding database. If the modification of the master file leaves the database in an inconsistent state the user will subsequently be prompted to edit the data in the corresponding database.

2.2.5. Special Purpose Components

Many application generators include some special purpose software packages which consist of functions that are useful in connection with the report generator and the graphics package to create very specialized applications more easily. Like the other modules of an AG, the special purpose software interfaces to the DBMS. The functions of this module are very often initiated via a menu, just like the data manipulation language.

We summarize the main features of two such packages which are most common in AG's for business applications. They are a software package for statistical analysis and another for financial modelling. Most application generators include some form of a statistical analysis package, which is especially useful for business forecasting applications. The spectrum of statistical tools covers means and standard deviation, correlation coefficients, analysis of variance, exponential smoothing and forecasting, and the like. To make the use of these tools more user-friendly they have been designed as interactive tools. The user selects the desired operation and then the system will automatically prompt him for the appropriate parameters. This makes the system more suitable for the casual user, for he does not have to memorize all the details.

The financial modelling language is an extension to the report generator with the purpose of creating financial statements, such as balance sheets or income and expense tables. It also supports the creation of financial models, for example projected capital needs and budget consolidation. The financial modelling language lets the user specify in an easy way how the particular columns of a report are to be computed. This is very much like the VisiCalc [16] package, except that, again, the financial modelling language of application generators interfaces with the data base management system. This enables the user to compute certain fields from data stored in the database.

3. A Generic Application Generator

In this section we will describe a generic application generator. This AG has essentially all the components described in the previous section. The syntax of this generic application generator has been chosen to be similar to the one of the AG's introduced before. We will now explain the following modules of the AG:

1. Database Management System
2. Report Generator
3. Data Manipulation Language
4. Graphics Package

3.1. Database Management System

As was pointed out before, application generators are based on a database management system (DBMS), which either employs a relational [3] or a hierarchical data model [8, 9]. In this presentation we have chosen a relational DBMS Tables (relations) are defined by the database administrator in the way shown in Fig. 3.1 below.

In this schema definition the three tables CUSTOMER, PRODUCT, and SALES are declared. For each relation (table) a key is specified, which consists of one or more fields (attributes). A field is specified by its name, its type, and its heading. In our generic application generator there are four built-in types, which are **number**(an integer), **money**(decimal number), **date**(in the form mm|dd|yy), and **text**(character string). The system can automatically extract the month, day, and year from attributes of type date. If, for example, the field CUSTADDR of the CUSTOMER table was used in a report the corresponding column would automatically be assigned the heading CUSTOMER,ADDRESS (the comma specifies the word ADDRESS to be written below CUSTOMER). The user can define virtual fields, i.e. pointers to fields of other tables. For example in Figure 3.1 the fields CUSTNAME of the SALES and the CUSTOMER tables are identical for those records that have a matching CUST # field.

```
schema

table CUSTOMER  key(CUST#)
        field CUST#          number   heading'CUSTOMER, number'
        field CUSTNAME       text     heading'CUSTOMER,NAME'
        field CUSTADDR       text     heading'CUSTOMER,ADDRESS'
end  table  definition

table PRODUCT  key(PROD#)
        field PROD#          number   heading'PRODUCT, number'
        field PRODNAME       text     heading'PRODUCT,NAME'
        field LSTPRICE       money    heading'LIST,PRICE'
        field UCOST          money    heading'PRODUCT,COST'
end  table  definition

table SALES  key(INVOICE#,PROD#)
        field INVOICE#       number   heading'INVOICE,NUMBER'
        field DATE           date     heading'SALE,DATE'
        field CUST#          number   heading'CUSTOMER,NUMBER'
        field PROD#          number   heading'PRODUCT,NUMBER'
        field UNITS          number   heading'UNITS,SOLD'
        define CUSTNAME  pointer CUSTNAME  in  table CUSTOMER key CUST#
        define CUSTADDR  pointer CUSTADDR  in  table CUSTOMER key CUST#
        define PRODNAME  pointer PRODNAME  in  table PRODUCT  key PROD#
        define LSTPRICE  pointer LSTPRICE  in  table PRODUCT  key PROD#
        define UCOST     pointer UCOST     in  table PRODUCT  key PROD#
end  table  definition
end  schema  definition
```

Figure 3.1: Database Definition

3.2. Report Generator

Let us consider a few examples of report generation over the database just defined. In Figure 3.2(b) a tabular report is created by the AG program shown in Figure 3.2(a). The report lists all the customers together with the respective number of units that were sold to them in the years 1980 through 1982. First the programmer has to specify the database on which the report is based, in this case the file SALES. Later on we will see an example of a program where two data bases are combined in one report using the join operator,which is a standard relational database operator, see Date [17]. Furthermore the programmer defines a title for the report. At the very end of the program he provides a screening condition, which, in this case, specifies the use of only those SALES records whose DATE field is between 1980 and 1982. Such a screening condition can be any

arbitrarily complex boolean expression over one or more fields of the database. In this example we have used the built-in functions sum, columntotal, and rowtotal. These built-in functions constitute the report generator special purpose commands which can be applied to any tabular report being generated.

```
report
file SALES
title 'UNITS SOLD PER CUSTOMER'
        list
          by CUSTOMER
          across YEAR
          sum(UNITS)
          rowtotal
          columntotal
        where YEAR  in 1980..1982
end  report
```

(a)

UNITS SOLD PER CUSTOMER

	YEAR 1980	1981	1982	
CUSTOMER	UNITS	UNITS	UNITS	TOTAL
COMP.DEVELOPMENT,LTD	23458	34563	43210	101231
ENGINEERING ASSOC.	5979	19820	9983	35782
BIGMONEY,INC.	98877	54438	78945	231260
BLACKMARKET,LTD	23451	9983	32564	65998
SOFTTEST ASSOC.	76590	65094	43679	185313
TOTAL	228355	183898	208381	619584

(b)

Figure 3.2: Program to Generate a Report and the Resulting Output

Figure 3.3 outlines the basic syntax of the report generator command. The user has to specify the file name(s) on which the report is based. Optionally a title can be specified. Possible verbs are list, print, sort, sum, etc. The objects listed after the by clause will appear in one column of the tabular report, those after the across clause create a new column for each different value, like YEAR in Figure 3.2(a). An object is either a field of the database or any function (or computation) applied to

one or more fields, for instance sum(UNITS) in Figure 3.2(a). The where clause provides for data screening such that only data records will be included in the report that fulfill the screening condition, which can be any boolean expression. The syntax of the screening condition is:

```
where <field-id> <bool-op> <test-values>
        {and/or <field-id> <bool-op> <test-values>}
```

The possible system functions that are usually provided are too numerous to list here, but they include formatting control, like page break specifications, and numerical functions, such as computing averages, regression analysis, or simply the calculation of column and row totals.

```
report
  file <FILE NAME>
 [ title <'TITLE TEXT'>]
 <VERB>
    by <OBJECT> [as <'TITLE'>] {<OBJECT> [as <'TITLE'>]}
    [across <OBJECT> [as <'TITLE'>] {<OBJECT> as <'TITLE'>}]
    [<SYSTEM FUNCTIONS>]
  [where <SCREENING CONDITION>]
end  report
```

Figure 3.3: The Syntax of the Report Generator Command

One might argue that the basic report generator statement as described in Figure 3.3 is not powerful enough when more complex calculations are asked for. For this reason most AG's have added some additional feature, such as the define feature which is shown in the program of Figure 3.4. This feature allows the user to temporarily define new fields (virtual fields) in the database and make use of these fields in the generation of the report. Actually we can view these fields as variables.

Thus in Figure 3.4(a) for each SALES record the new fields DPRICE, NCOST, PROFIT, DPROFIT, and DIFF are computed and temporarily stored (the value of these fields depends on the number of units sold, if more than 200 units a special discount of 10% is granted). Note that the types of these fields are extracted from the type definition in the database, i. e. the user is not concerned about this. For example the system automatically determines the type of the new field DPRICE to be the same as for the field LSTPRICE, namely money. The basic syntax of the define statement is as follows:

```
define
file SALES
  if UNITS  gt 200  then  do
        DPRICE=LSTPRICE
        NCOST=UCOST
  end
  else  do
        DPRICE=LSTPRICE * 0.9
        NCOST=UCOST * 0.85
  end
 PROFIT=(LSTPRICE-UCOST) * UNITS
 DPROFIT=(DPRICE-NCOST) * UNITS
 DIFF=DPROFIT-PROFIT
end  define

report
file SALES
title'UNDISCOUNTED VERSUS DISCOUNTED PROFIT'
  list
     by YEAR
     sum(UNITS)  as 'TOTAL,UNITS'
    PROFIT  as 'UNDISCOUNTED,PROFIT'
    DPROFIT  as 'DISCOUNTED'PROFIT'
    DIFF  as 'DIFFERENCE'
  where YEAR  in 1980..1981
end  report
```

(a)

UNDISCOUNTED VERSUS DISCOUNTED PROFIT

YEAR	TOTAL UNITS	UNDISCOUNTED PROFIT	DISCOUNTED PROFIT	DIFFERENCE
1980	62412	2,255,587.52	2,907,591.64	652,004.12
1981	87098	2,824,039.17	3,634,877.83	810,838.66

(b)

Figure 3.4: Report Involving Some More Complex Calculation

```
define
file <FILE NAME>
   {if <SCREENING CONDITION> then <DEFINITION OF NEW FIELD(S)>
                             else <DEFINITION OF NEW FIELD(S)>}
   [<DEFINITION OF ADDITIONAL FIELD(S)>]
end define
```

Note that the define program in Figure 3.4(a) contains an implicit loop since the calculation is

performed for each record in the database. This is one of the main differences between this program and a program written in a conventional programming language. In the subsequent report generation phase the newly defined fields can be used just like the fields physically stored in the database. In this case the discounted profit (DPROFIT) is compared with the undiscounted profit (PROFIT) and the difference (DIFF) is listed for the years 1980 and 1981.

Earlier in this paper it was stated that report generators are non-procedural. This, however, does not mean that they cannot have subroutines. In Figure 3.5 we see an example of a parameterized procedure. In this example the last two digits of the year and the PNAME (for PRODNAME) are parameters. The program computes the total number of units sold of the product specified in the parameter PNAME during the year YR. Note that the syntax requires us to prefix the formal parameters by an ampersand whenever it is used in the program.

```
procedure SALES_OF_PROD
arguments YR:yy,PNAME: text
file SALES
title 'TOTAL UNITS SOLD OF &PNAME  in 19&YR'
        list
          sum(UNITS)  as 'TOTAL UNITS SOLD'
        where YEAR  is &YR
              and PRODNAME  is &PNAME
end  procedure
```

This procedure can be invoked as follows:

```
exec SALES_OF_PROD (YR=>79,PNAME=>WORKSTATION)
```

Figure 3.5: Example of a Parameterized Procedure

Now let us consider an example of a report generation that combines two databases. For this purpose in Figure 3.6 two more database tables are defined: STUDENT and PROFESSOR. Let us assume the user wants to list all professors of the CSCI department and the average GPA of their advisees. This asks for a join of the two tables PROFESSOR and STUDENT. In Figure 3.7 these two tables are joined on the fields ADVISOR and PROF_NAME. Then the program generates a listing of all professors in the CSCI department together with the average GPA of their students. The syntax of

the join operator is outlined as

```
join files (<FILE NAME>,<FILE NAME>) matching (<FIELD>,<FIELD>)
                                      {matching (<FIELD>,<FIELD>)}
```

Note that we can actually join two databases on more than one field. This means that the respective data records are only combined if they agree on all fields specified in the **matching** statement.

```
schema

table STUDENT  key(STUD_NAME)
        field STUD_NAME      text        heading 'STUDENT,NAME'
        field ADVISOR        text        heading 'ADVISOR'
        field GPA            number       heading 'GRADE POINT,AVERAGE'
end  table  definition

table PROFESSOR  key(PROF_NAME)
        field PROF_NAME      text        heading 'PROFESSOR,NAME'
        field DEPT           text        heading 'DEPARTMENT'
end  table  definition

end  schema  definition
```

Figure 3.6: Schema Definition of Tables STUDENT and PROFESSOR

```
report
file STUDENT
file PROFESSOR
join FILES (STUDENT,PROFESSOR)  matching (ADVISOR,PROF_NAME)
        list
          by PROF_NAME
         AVE(GPA)  as 'AVERAGE,GPA'
        where DEPT  is 'CSCI'
end  report
```

(a)

```
PROFESSOR            AVERAGE GPA
NAME               OF HIS/HER STUDENTS
-----------------------------------
PROF. EASYGOING          3.95
PROF. SMITH              3.25
PROF. TOUGHMAN           2.55
  ..                     ..
  ..                     ..
```

(b)

Figure 3.7: Report Generation Using Two Tables

3.3. Data Manipulation Language

In this section we will investigate another feature of application generators:the data manipulation language. This language is used to maintain the data stored in the database. It enables the user to perform the following transactions:

- insert data
- update data
- delete data
- perform integrity checks on the data

The syntax of the data manipulation language follows closely the syntax of the report generator as far as this is possible, so that the user has to memorize as few concepts as possible. To modify a database the user has to write a short program as illustrated in Figure 3.8. Then he has to provide the data for this program. In our example we want to modify the database SALES. The program will take as input the INVOICE# and the PROD# and compare whether a record with these field values is already in the database. If so it updates the other fields according to the data provided in the data entry section (this is specified in the statements following **on match do**. If the record is not found in the database it will be inserted and the fields get the values defined in the data entry section.

The basic syntax of the data manipulation language is outlined in Figure 3.9. As usual, the first thing the user has to specify is the file he wants to modify. Then he has to specify the fields of this database that he wants to match with the entered data in the data entry section. Then the code for the case that the entered data matches some records in the database is given. This code is placed after the key words **on match do**. This section can contain commands to **update, delete, or insert** data records. Following this the code for the case that none of the records in the database matches the entry is defined. This is indicated by the key word **on nomatch do**.

```
modify
file SALES
  match INVOICE#,PROD#
  on match do
     update
        CUST#,UNITS,DATE
  on nomatch do
     include
end modify

Now the user has to provide the data, i.e. the fields INVOICE#,PROD#,
CUST#,UNITS, and DATE:

begin data
INVOICE#=1023
PROD#=132
CUST#=12
UNITS=200
DATE=11/30/79

INVOICE#=1024
  ..
  ..
  ..
end data
```

Figure 3.8: Modification of the SALES Database

```
modify
file <FILE NAME>
  match <FIELD> {<FIELD>}
  on match do
    <CODE FOR THE MATCHING CASE>
  on nomatch do
    <CODE FOR THE NONMATCHING CASE>
end modify

begin data

<DATA ENTRY SECTION>

end data
```

Figure 3.9: Outline of the Syntax of the Modify Statement

3.4. Graphics Package

Another way to generate reports in an application generator is in the form of graphs. In this section 'graphical report generation' refers to report generation using the graphics package as distinct from 'tabular report generation' described in the Section 3.2. We will take a look at examples of graphical reports, the language for graphical report generation, issues involved in their design and reasons for their effectiveness as a programming tool.

Graphical reports can be of one of the following types.

- histograms
- piecharts
- barcharts
- curve plots
- scatter diagrams

All the features of the report generator, such as performing arithmetic, screening data etc. can be used in graphical report generation. Thus one is able to produce reports having a greater visual content with almost the same ease.

The following examples illustrate graphical report generation. The schema definition of the database used in the examples in this section is given in Figure 3.10.

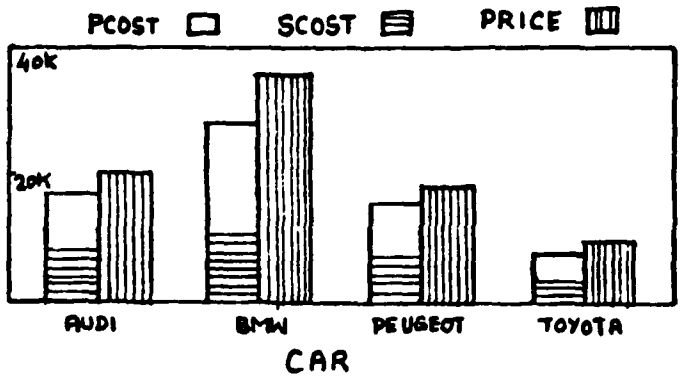
```
schema

table CARS  key(CAR,COUNTRY)
        field CAR          text      heading'NAME OF,CAR'
        field COUNTRY      text      heading'COUNTRY'
        field PCOST        money      heading'PRODUCTION,COST'
        field SCOST        money      heading'SALES,OVERHEAD'
        field SALES        number     heading'NUMBER,SOLD'
        field PRICE        money      heading'PRICE'
        field MPG          number     heading'MILEAGE'
end  table  definition
end  schema  definition
```

Figure 3.10: Schema Definition of Table CARS

The request in Figure 3.11 is a report in the form of a histogram of the total cost and price of the various cars. Note that the over feature enables us to split the total cost into its components and have them displayed one over the other in different shades; the and feature enables us to compare the cost and price of each car by producing adjacent bars.

```
histogram
  file CARS
  draw
  PCOST  over SCOST  and  PRICE
  across CAR
  where SALES  gt 2000
end
```

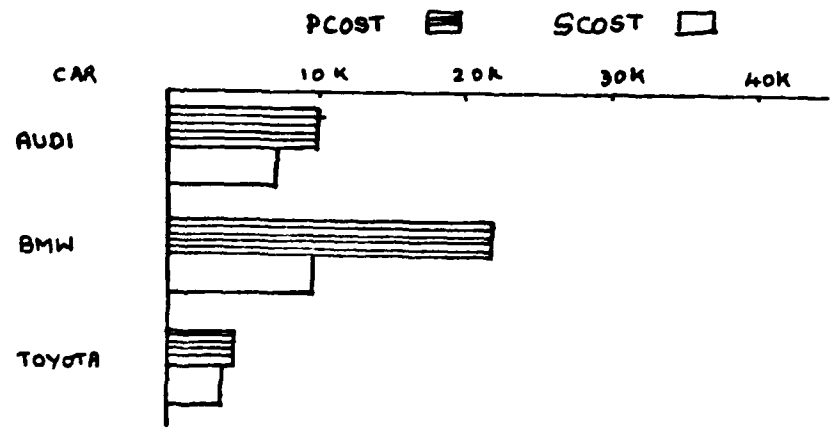
Figure 3.11: Histogram



The program in Figure 3.12 generates a barchart of the production cost and sales overhead of the different cars. The use of the where clause screens the data, such that only cars of the specified three types are included.

```
barchart
  file CARS
  draw
  PCOST  and SCOST
  across CAR
  where CAR="AUDI"
     or CAR="BMW"
     or CAR="TOYOTA"
end  barchart
```

Figure 3.12: Barchart



In Figure 3.13 a curve of the retail cost, dealer cost, and rpm (revolutions per minute) of the cars is plotted against their fuel consumption.

In Figure 3.14 a piechart of the fuel consumption of the various cars is shown.

Now let us take a closer look at the language of our programs. The basic syntax for graphical report generation is defined in Figure 3.15.

```
define
  file CARS
  TCOST=PCOST+SCOST
end define
curve
  file CARS
  draw
  TCOST  and PRICE
  across MPG
end  curve
```

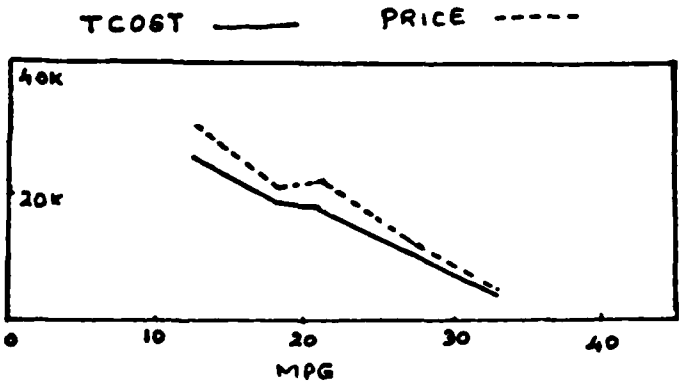


Figure 3.13: Curve Plot

```
piechart
  file CARS
  draw
 MPG
  across CAR
end
```

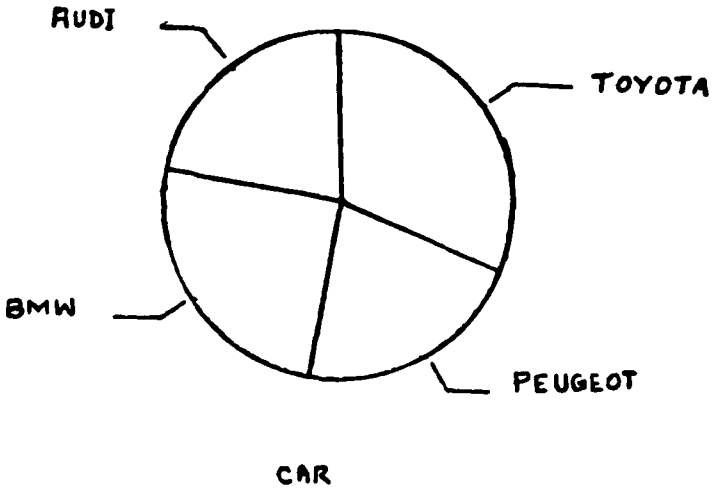


Figure 3.14: Piechart

```
<definition part>
<TYPE OF GRAPH>
   file <FILENAME>
   [ <options> ]
   draw
  <DEPENDENT OBJECT> { over <DEPENDENT OBJECT>}
     { and <DEPENDENT OBJECT> { over <DEPENDENT OBJECT>}}
   across <INDEPENDENT OBJECT>
   where <SCREENING CONDITION>
 end <TYPE OF GRAPH>

<DEPENDENT OBJECT> ::= <OBJ.NAME> [<OBJ.TITLE>] [<OBJ.SHADE>] [<OBJ.SCALE>]

<INDEPENDENT OBJECT> ::= <OBJ.NAME> [<TITLE>] [<OBJ.SCALE>]
```

Figure 3.15: Syntax of Programs Using a Graphics Package

<TYPE OF GRAPH> can be histogram, piechart, barchart, scatter diagram, or curve. <DEPENDENT OBJECT> refers to the attribute whose value is to be plotted in the graph. <INDEPENDENT OBJECT> is the one against which the other attributes are to be plotted. Note the similarity of this syntax and the syntax of tabular report generation. This similarity has been created deliberately to aid the users in learning and using the application generators.

In addition to what is described above, it is also possible to supply optional information to the graphics package in the form of specific titles and shades for the bars in the histograms (or barcharts), types of lines to be used in the curve plots, and the display area in which the report is to be generated.

Now let us consider three very important characteristics of application generators. First, as in the case of tabular report generators, the user need not concern himself with details of his data and its organization and retrieval. One only needs to specify the data that one wants to see plotted. Secondly, it suffices to specify the desired graph at a very high level. The user does not have to "program" starting from graphical primitives. Finally the application generators provide a smooth integration of report generation and graphics; without this integration it would be the responsibility of the user to combine the two aspects of the system in his programs explicitly.

These three characteristics give the graphics packages of report generators considerable power and create a simple and convenient medium, so the user can communicate his requirements to the computer. However, we should not fail to note the limitations of these packages. They are not general purpose graphics packages; they derive their power precisely by not being general and being limited to a certain domain.

4. Can an AG be used for General Purpose Programming?

In this section we examine the characteristics of application generators which distinguish them from programming languages. Then we analyse the possibility of combining the high level features of application generators with general purpose programming languages.

4.1. Comparison of AG's with General Purpose Programming Languages

Application generators are typically used in the context of a database management system. The applications normally do not entail the use of explicit iteration or recursion in the programs. Most often, they only involve applying the same operation on all records of a database (or all records which

obey a set of screening conditions). Hence, the application generators do not offer looping constructs. For the same reason, application generators do not provide data structuring facilities or typing mechanisms. They only make use of the data types occurring in the database involved in the application.

A tabular comparison of programming languages and application generators is provided in Figure 4.1.

Programming Languages	Application Generators
procedural; programs define computations step by step	non-procedural; very high level programs that state the required results
usually non-data-intensive	data-intensive
explicit iteration and loops	implicit iteration
explicit typing mechanisms	implicit typing mechanisms
very wide range of applications	limited problem domains
detailed documentation necessary	mostly self-documenting
prototyping is usually slow and errorprone	supports fast and correct prototyping
difficult to maintain	easier to maintain

Figure 4.1: A Tabular Comparison of Application Generators With General Purpose Programming Languages

In general we can say that programs written using an AG are very high level statements of the required results. Hence those programs can be written faster, are self-documenting, and are easier to maintain. On the other hand general purpose programming languages have more computational flexibility than application generators. Programming languages gain this flexibility over application generators by having explicit iteration constructs, data structures, such as arrays, records, pointers, etc., and explicit typing mechanisms. All these features are presently missing in application

generators.

Contemporary application generators typically provide an interface from a general purpose programming language to the database management system and to some of the high level AG commands. Graphically this interface can be described as in Fig. 4.2.

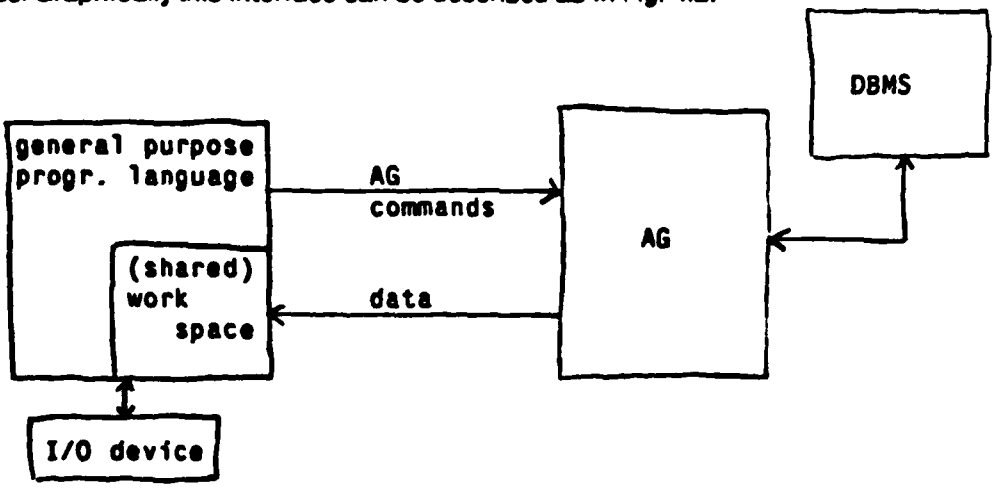


Figure 4.2: Organization of a Typical Interface Between a General Purpose Programming Language and an AG

Fig. 4.2 shows that in those systems the application generator and the general purpose programming language can be viewed as two separate components. If the user wants to use some AG routine from within his program he has to create a (shared) work space in his program first and then issue the AG command in the form of an external procedure call. Only the application generator has access to the DBMS. The AG will store the resulting data in the user-provided work space, where it can then be accessed from within the general purpose programming language program.

The problem with this organization is that the use of the application generator commands is very tedious. The user has to provide a workspace in form of records, arrays, or the like before he can execute the application generator commands. The application generator returns data and stores it in the predefined workspace, from which the user can then retrieve it for processing. Since there is no real embedding of the application generator features in the general purpose programming language

the user still has to explicitly iterate over the data stored in the workspace and retrieve each record individually from there. Also the user has to know in advance the size of the resulting data in order to reserve a sufficiently large work space.

4.2. Embedding Higher-Level Data Retrieval Constructs in Programming Languages

A more elegant way of designing languages for data intensive applications was followed in the design of embedded data manipulation languages in general purpose programming languages, which were summarized by Stonebraker and Rowe [18]. Examples of these are EQUEL [19], which is an embedding of the database query language QUEL in the programming language C, THESEUS [20], which embeds relational operators in the language Euclid. Schmidt [21] reports some work on embedding relational constructs in Pascal by describing tuples essentially as Pascal records. Similar work was done by Wasserman et al [22] in the form of integrating relational operators in a Pascal-like language. But none of these efforts has gone as far as extending the high level application generator facilities to their system.

To get a better idea of these systems let us look at an example program written in PLAIN [23]. The program in Fig. 4.3 makes use of the external database tables CUSTOMER and SALES, similarly defined as in section 3.

```
program NewYearsPresent;

external readonly   {declaration of external relations}

        CUSTOMER:relation[key CUST#] of
                CUST#      :integer;
                CUSTNAME   :string;
                CUSTADDR   :string;
        end CUSTOMER;

        SALES:relation[key INVOICE#,PROD#] of
                INVOICE#   :integer;
                DATE       :string;
                CUST#      :integer;
                UNITS      :integer;
                LSTPRICE   :float;
        end SALES;

end external;
```

```
procedure ComputeBonus;
imports CUSTOMER,SALES:readonly;   {import of global var.}

var

        Good_Customer:relation[key id] of  {internal relation}
                id      :integer;
                total   :float;
                name    :string;
                address:string;
        end Good_Customer;

        {In the following lines we declare two markings over SALES,CUSTMER}
        cust:marking of CUSTOMER (CUST#,CUSTNAME,CUSTADDR);
        tempsales:marking of SALES (INVOICE#,PROD#,UNITS,LSTPRICE);
        total,bonus:float;

        begin
        Good_Customer:=[];
        cust:=CUSTOMER where CUSTADDR="*Los Angeles*"  {select tuples and}
                =>(CUST#,CUSTNAME,CUSTADDR);           {project on 3 attr.}

        foreach c in cust  { loop over all tuples in cust }
        loop <byCustomer>
                tempsales:=SALES where CUST#=c.CUST# and
                        DATE="*1982*" =>(INVOICE#,PROD#,UNITS,LSTPRICE);
                total:=0;

                foreach t in tempsales  { loop over all tuples }
                loop <bySales>
                        total:=total+(t.UNITS*t.LSTPRICE);
                repeat <bySales>;

                if total > 10000 then { add him to Good-Customer }
            Good_Customer:+[<c.CUST#,total,c.CUSTNAME,c.CUSTADDR>];
        repeat <byCustomer>;

        foreach gc in Good_Customer { loop over all tuples
        loop <byGood_Customer>
                bonus:=gc.total/100;                    {generate listing of }
                writeln(gc.name,gc.address,bonus);      {all "good" customers}
        repeat <byGood_Customer>;

end ComputeBonus;

        Computebonus;   {call the procedure}

end NewYearsPresent.
```

Figure 4.3: PLAIN Program to Compute the 1% Discount for all Customers in L.A. Who Bought Goods for More Than $10,000 in 1982

The PLAIN program in Fig. 4.3 makes use of two external relations (e.g. SALES and CUSTOMER) and determines all those customers in L.A. that have bought goods for more than $10,000 during the year 1982. To do this one additional internal relation, Good_Customer, is declared, to which all the "good" CUSTOMER-tuples are added. Furthermore two markings are declared in the program. A marking is just a selection of certain tuples of the relation over which the marking was declared. For more details we would like to refer the reader to Van De Riet [23]. The program structure is such that for each customer in L.A., stored in the CUSTOMER relation, the program loops through the SALES relation and sums up the value of their purchases, given by LSTPRICE*UNITS. If this totals to more than $10,000 the particular customer is added to the relation Good_Customer and gets printed out at the end.

From the AG viewpoint the deficiencies of this program would be:

- tedious explicit iteration over all the records in the relation (or marking)
- low level computational details have to be specified; for example initializing the variable total to zero in the above program
- high overhead of type and variable declarations, which are actually implicit in the database definition

None of these deficiencies would be present in an AG based program. The above listed points are common to all existing embedded database languages and therefore make these systems hard to use for non-programmers.

One way of extending the power of application generators would be by following the same approach as in embedding data manipulation languages in general purpose programming languages. This approach consists of embedding the high level features of AG's in a programming language, thereby achieving both, the ease of use of application generators as well as the computational flexibility of a conventional programming language. In such a hypothetical system the problem of determining the "good" customers could be solved as outlined in Fig. 4.4.

In this example (Fig. 4.4) we first compute the total field for each CUSTOMER tuple by implicitly

```
file CUSTOMER

        define
          total=
                { file SALES
                    sum by CUSTOMER.CUST#
                        LSTPRICE*UNITS
                    where DATE in 1982
                  end file SALES }

          bonus=total/100
        end define


        report
                list
                        CUSTNAME,CUSTADDR,total,bonus
                where total > 10000
                      and CUSTADDR in "Los Angeles"
        end report

end file CUSTOMER.
```

Figure 4.4: Hypothetical AG Program for the "Good Customer" Problem

looping through the SALES database and summing up LSTPRICE*UNITS for identical CUST#'s. Then the bonus is computed, again for each tuple in the CUSTOMER database, as 1% of the total. Finally a report is generated where all the customers in L.A. whose total exceeds $10,000 are selected. Comparing this with the PLAIN program in Fig. 4.3 we observe that it is definitely much shorter and easier to understand since only the most relevant steps of the computation are specified. However it is possible that the AG program will be less efficient.

4.3. Future Research

For the purpose of this paper we have sought to explain to the software engineering community why application generators have proven so useful in the "edp world". As a future research area we see the integration of application generator features in general purpose programming languages as a potentially powerful way to improve programmer productivity. This is the goal of our current research. In a later paper we plan to give a more detailed presentation of such a system.

References

1. Munson,J.B. and Yeh,R.T., "Report by the IEEE Software Productivity Workshop," San Diego,Ca., 1981.

2. Boehm,B.W., "Software and its impact: A quantitative assessment," *Datamation*, May 1973, .

3. McCracken,D.D., *A Guide To Nomad For Application Development*, National CSS, 1980.

4. Martin,J., *Application Development Without Programmers*, Prentice Hall,Inc., Englewood Cliffs, New Jersey, 1982.

5. Druffel,L., *Strategy for a DoD Software Initiative*, Dept. of Defense, August 27, 1982, Volume 1 and 2

6. Tucker,A.B., *Programming Languages*, Mc Graw-Hill, Computer Science Series, 1977.

7. National CSS, "NOMAD 2: Reference Manual," National CSS, Wilton, CT 06897, 1982.

8. Information Builders, Inc, "FOCUS Users Manual," Information Builders, Inc., 1250 Broadway, New York, N.Y.10001, 1982.

9. MPG, "RAMIS 2: Users manual," Mathematica Products Group, Inc., Princeton, N.J. 08540, 1982.

10. Johnson,R.C., "Automated Software Development Eliminates Application Programming," *Electronics*, June 1982, .

11. Ashton-Tate, "dBASE II Users Manual," Ashton-Tate, 3600 Wilshire, Los Angeles, Ca. 90010, 1981.

12. IBM, "IMS Application Development Facility (IMSADF)," IBM General Information Manual GB21-9869-2, IBM Corporation, East Irving, Texas 75062, 1980.

13. Leavenworth,B.M. and Sammet,Jean E., "An Overview of Nonprocedural Languages," *Proceedings of a Symposium on Very High Level Languages*, Sigplan Notices, 1974.

14. Leavenworth,B.M., "Non-Procedural Data Processing," *The Computer Journal*, Jan 1976, .

15. Zloof,M.M., "QBE/OBE: a language for office and business automation," *IEEE Computer*, Vol. 14, No. 5, May 1981, pp. 13-23.

16. Williams,R.E. and King.B.L., *The Power of VisiCalc*, Management Information Source, 1982.

17. Date,C.J., *An Introduction to Database Systems*, Addison-Wesley Publishing Company, 1977.

18. Stonebraker,M. and Rowe.L., "Observations on Data Manipulation Languages and Their Embedding in General Purpose Programming Languages," *Proceedings of the Third International Conference on Very Large Data Bases*, 1977, .

19. Stonebraker et al, "The Design and Implementation of INGRES," *ACM-TODS*, Vol. 1. No. 3. 1976, .

20. Shopiro,J.E., "THESEUS-A Programming Language for Relational Databases," *ACM-TODS*, Vol. 4, No. 4, December 1979, pp. 493-517.

21. Schmidt,J.W., "Some High Level Language Constructs for Data of Type Relation," *ACM-TODS*, Vol. 2, No. 3, Sept 1977, .

22. Wasserman, A.I. et al, "Revised Report on the Programming Language PLAIN," *ACM-SIGPLAN*, May 1981, .

23. Van De Riet,R.P., Wasserman,A.I., Kersten,M.L., and De Jonge,W., "High Level Programming Features for Improving the Efficiency of a Relational Database System," *ACM-TODS*, Vol. 6, No. 3, Sep 1981, .

24. Codd,E.F., "The 1981 Turing Award Lecture. Relational Database:A Practical Foundation for Productivity," *CACM*, Vol. 25, No. 2, Feb 1982, .

25. Schmidt,J.W., "Type Concepts for Database Definition," *Proc. Int. Conf. on Databases:Improving Usability and Responsiveness*, Academic Press,New York, 1978, pp. 215-244.

26. Prywes,N.S., Pnueli,A., and Shastry,S., "Use of a Nonprocedural Specification Language and Associated Program Generator in Software Development," *ACM Trans. on Progr. Lang. and Systems*, Vol. 1, No. 2, October 1979, pp. 196-217.

27. M.Hammer,W.Howe,V.Kruskal,and I.Wladawsky, "A Very High Level Programming Language for Data Processing Applications," *CACM*, Vol. 20, No. 11, Nov 1977, pp. 832-840.

28. *Symposium on Very High Level Languages*, SIGPLAN Notices, ACM, april 1974.

29. Zloof,M. and deJong,S., "The system for business automation (SBA): programming language," *CACM*, Vol. 20, No. 6, June 1977, pp. 385-395.

30. Wasserman,A.I., "A Software Engineering View of Data Base Management," *Proceedings of the Fourth International Conference on Very Large Data Bases*, 1978, .

31. Kersten,M.L. and Wasserman,A.I., "The Architecture of the PLAIN Data Base Handler," *Software-Practice and Experience*, Vol. 11, Feb 1981, .

32. Grochow,J.M., "Application Generators: An Introduction," *Proceedings of the 1982 National Computer Conference*, AFIPS, Houston, Texas, Jun 1982.

33. Waldrop,J.H., "Application Generators: A Case Study," *Proceedings of the 1982 National Computer Conference*, AFIPS, Houston, Texas, Jun 1982.

34. Cardenas,A.F. and Grafton,W.P., "Challenges and Requirements for New Application Generators," *Proceedings of the 1982 National Computer Conference*, AFIPS, Houston, Texas, June 1982.

35. Goodman,A., "Application Generators at IBM," *Proceedings of the 1982 National Computer Conference*, AFIPS, Houston, Texas, Jun 1982.