AUTOMATIC PROGRAMMING

Patricia C. Goldberg
Computer Sciences Department
IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

ABSTRACT: Work in Automatic Programming focusses on the problem of how to make the computer more accessible to the non-programmer. It involves both the design of interfaces and the development of algorithms to support those interfaces. It is also concerned with how to write programs which can be manipulated: changed, subsetted, or composed. This paper outlines one facit of the work in Automatic Programming at the IBM Research Center, Yorktown Heights, New York.

I. Introduction

The last few years have seen the formation of a number of groups working in the general area of Automatic Programming [1,2,3]. Although this term is used to identify a variety of research activities, these generally have in common an interest in the computer as a tool for end users who have little interest in programming as a skill. This is in contrast to the common goal of computer science research which focusses on the use of computers by computing professionals.

The aim of work in Automatic Programming is not to develop specific applications, but rather to develop application methodologies: that is, methods of specifying programs and systems of programs which will automate a given area. Because of the intended audience for these methodologies, there has been much attention paid to both very high level programming interfaces and "non-programming" interfaces [2,4,5].

The group at the IBM Research Center is primarily focussing on the area of business applications. There are several reasons for this. First, there is a fair amount of structure in business applications which can be exploited. Further, many small businesses could employ computers to advantage were it not for the cost of programming. Finally, because there is also a great deal of individual variation in businesses, we must face directly the problem of handling these idiosyncracies.

We are looking at many possible methods for specifying business programs. Some of these involve new programming interfaces in which the user directly specifies the application, while others concentrate on methods for determining what the user needs in the way of application programs and for composing the required programs out of program fragments. We are also looking at techniques for programming by example, a technique which would permit the user to specify a procedure by giving examples of the input-output relationship. Some of our work in this area involves design of appropriate interfaces, while other groups are concentrating on the development of efficient implementation techniques for realizing these interfaces.

In spite of the concern for methodologies for use in a programmerless environment, the Yorktown group is nonetheless very concerned with programs. In many respects the involvement with programs is intensified, by this point of view since progress must be considered as automatically constructible, manipulable objects. This leads to a somewhat different perspective on programs from that derived from a view of programs as manually constructed objects for one-situation use. In order to illustrate this point, and in order to show how some of the apparently disparate efforts in Automatic Programming fit together, the following sections elaborates one problem area and our solution to it.

II. The Problem

One approach to the problem of developing tailored applications in a programmerless environment is to write a large number of program fragments which, taken as a whole, represent all of the common combinations of performing a given business function. Then, on the basis of information received from the ultimate user concerning his particular application needs, these program fragments are subsetted, modified, and composed to build the required application programs.

Such a view of application development raises several interesting questions. How ought program fragments be expressed to facilitate the kind of manipulations required? Can such program fragments be written by an expert familiar with the application area rather than by someone who is an expert in data processing? How are programs truly peculiar to a given application to be introduced? Is it possible to combine the programs written for two separate areas to form one application package?

In the process of eliciting information from the end user concerning his particular requirements, what kind of user-computer dialogue is necessary and how do we relate the information gathered from the user to the program fragments? What models of the programs and computing processes,

other than the programs themselves, are required in developing such a system? What other models of the business world are useful? How do we correlate these with the information received with the user?

Finally, how do we reconcile the requirements imposed on the programs from the standpoint of manipulability with the requirement that they make reasonable use of machine resources? How do we decide on appropriate data representations? What kind of analysis and optimization techniques are peculiarly required by programs so generated?

The construction of a feasible system for generating applications from prestored program fragments, requires that each of these questions must be faced directly; otherwise the resulting system will be nothing but a toy. Each of these facits of the problem are being addressed by the Yorktown Automatic Programming group. The bulk of the discussion below, however, is concerned with the specification of a programming interface in which to develop the program fragments. The work on the other issues raised above is briefly discussed in Section V.

III. The Programming Interface

Is a new programming interface required for this kind of work? The following example (one to which we shall return frequently) illustrates some of the problems with existing languages.

Consider the task of taking a set of incoming orders and producing the invoices for the goods ordered. Assume that it has been decided to produce separate invoices for each order. In this case the program, in a conventional language such as PL/I, is rather trivial. That is, it consists of a looping construction which iterates over all the elements of the input file and for each element in the input file produces an output, which is essentially a copy of the input with additional calculations. This code is shown in figure 1. It is assumed in this example that the two files required, CUST_MAST and ITEM_MAST, are small enough to be represented as simple v in primary storage. Were this not the case, decisions concerning the appropriate I/O statements and access methods would have to be incorporated into the code -- further complicating the situation.

```
DO I = 1 TO NO_ORDERS;

   INVOICE.NAME = CUST_MAST(ORDER(I).CUST#).NAME;
   INVOICE.ADDR = CUST_MAST(ORDER(I).CUST#).ADDR;
   INVOICE.CUST# = ORDER(I).CUST#;
   INVOICE.GROSS = 0;

   DO J = 1 TO ORDER(I).NO_ITEMS;

      INVOICE.ITEMS(J).ITEM# = ORDER(I).ITEMS(J).ITEM#;
      INVOICE.ITEMS(J).QUANT = ORDER(I).ITEMS(J).QUANT;
      INVOICE.ITEMS(J).PRICE =
        ITEM_MAST(INVOICE.ITEMS(J).ITEM#).PRICE;
      INVOICE.ITEMS(J).EXTEND = INVOICE.ITEMS(J).PRICE *
                                INVOICE.ITEMS(J).QUANT;
      INVOICE.GROSS = INVOICE.GROSS +
                      INVOICE.ITEMS(J).PRICE;
      IF INVOICE.ADDR.STATE = 'NEW YORK'
        THEN
        INVOICE.TAX = .05 * INVOICE.GROSS;
        ELSE
        INVOICE.TAX = 0;
      INVOICE.AMOUNT = INVOICE.GROSS + INVOICE.TAX;
   END;

   CALL PRETTY_PRINT(INVOICE);

END;
```

FIGURE 1. CODE TO PRODUCE INVOICES FROM ORDERS -- ONE PER ORDER

The program in figure 2 is for the same problem, except that it produces only one invoice for each distinct customer who has placed an order during that period. This, of course, implies that several orders may be collected to produce one invoice. In spite of what appears to be a rather minor change in the specification, the PL/I program for this function looks very different from the original program. It still begins with a loop over the input documents, but in this case the loop is used only to sort the orders into piles of orders with common customer name. There is necessarily a second loop which takes each pile and produces an invoice. This includes logic to insure that, if the same item is contained on several orders for the same customer, only one mention of the item appears on the invoice along with the appropriate adjustment to the quantity of that item that has been ordered.

```
J = 0
DO I - 1 TO NO_ORDERS;
   DO K = 1 TO J;

      IF ORDER(I).CUST# = M_ORD(K).CUST#
         THEN CALL MERGE(ORDER(I).M_ORD(K));
          /* MERGE adds in new items to M_ORD and
             adjusts the quantity for previously
             ordered items */

   END;

   IF K = J + 1 THEN
      DO;
        J = J + 1;
        M_ORD(J).CUST# = ORDER(I).CUST#;
        DO L = 1 TO ORDER(I).NO_ITEMS;
          M_ORD(J).ITEMS(L).ITEM# = ORDER(I).ITEMS(L).ITEM;
          M_ORD(J).ITEMS(L).QUANT = ORDER(I).ITEMS(L).QUANT;
        END;

      END;

   END;

   DO I = 1 TO J;

        .
        .  /* Prepare Invoice - Similar to Fig. 1  */
        .

      CALL PRETTY_PRINT (INVOICE);

   END;
```

FIGURE 2. CODE TO PRODUCE INVOICES FROM ORDERS -- ONE PER CUSTOMER

Even though the *specifications* for these two programs differ only slightly, the programs vary in nontrivial ways, so that it would be difficult to see how to generate one from the other in a simple manner. It would probably be easier to write a new program than to attempt to modify the first. It can be argued that this is the result of a "hidden "optimization" in the first program which causes calculations for the preparation and aggregating of the input to be merged with those for producing the output documents.

The function originally described above is clearly a batch processing function in the sense that it takes a predetermined *set* of orders and produces the *set* of invoices. Suppose now that we wanted to place essentially the same function in an on-line environment, in which case we would want the module that produces orders to send them individually, as they are created, to the module that produces invoices. Although in this case the change to the *invoice* producing module would not be catastrophic, the effect on the overall system structure and the order in which modules make calls on other modules, would be. Yet again, conceptually, from an applications point of view, the change is trivial.

There are other objections which we can make to writing these programs in PL/I or any other computer oriented language. For example, the model writer must distinguish between the same function when applied to large files and when applied to small files. This largely because of the difference in accessing techniques for primary and secondary storage devices. Furthermore the programmer must be concerned with the various formats of the data, specifically the difference between internal and external representations. In many languages these distinctions, as well as issues concerned with word length, etc., are spread throughout the code. All of these problems make it necessary to find *in the same individual* both enough application knowledge and data processing skill to write the required program fragments and the rules of composition.

From these comments it is possible to construct a list of requirements which the programming interface must at least meet if it is to prove suitable for this task. First, it must be a very high level interface in which data processing notions are suppressed and terms, concepts and constructs reasonable to the application expert are utilized. This requirement stems simply from the impossibility of finding people skilled in both areas.

There are other, more technical, characteristics that the language should also have which influence how these high level notations are introduced. One would like to guarantee that the specifications for an application option map into some relatively local fragment This, of course allows us to modify a program for one application option to handle a different option by making trivial local changes to the program.

There is another technical reason for this requirement. It is highly desirable to program the fragments in such a way that application options that are apparently independent do not affect the

same pieces of code. Put another way, dependencies that are not present in the application itself should not appear in the program representing the application. Finally, the language should place minimal fixed sequencing constraints on the program. Such a language makes it easier to insert and remove functions. It will also make it easier to support batch and data entry functions for the same programming module.

IV. The Business Definition Language

The Business Defintion Language, BDL [5], has been defined with these criteria in mind. It is aimed at a particular set of business applications, namely, the paper handling, data manipulation functions. More specifically it is directed towards applications which are minimally involved with computation and maximally involved with data manipulation. It is almost certainly not the language that should be used in writing, for example, the linear programming algorithms used in warehouse utilization problems.

The language itself is highly structured and is designed to support a particular style of programming. It is intended that for any function there will be one obvious way to program it. That is, there should not be a number of possible computational paths to the solution. Furthermore, the one path should be obvious to someone skilled in the application area. In particular, the application programmer will not be concerned with issues of efficiency nor will there be choices in the language which will allow him to substantially effect the efficiency of the algorithm. Rather, the language has been designed so that the structure of the programs will reflect the structure of the business application.

The view of business that is reflected in this programming language is that a business consists of operating entities, such as departments or sections, or clerks; and that these operating entities communicate with one another largely on forms or documents. The function of these entities, at least the parts and that can be reasonably automated, is to transform their incoming forms to the kind of forms that they produce.

As a consequence of this view of business the BDL language contains as primary elements:

steps, which are used to represent the operating entities;

documents, which represent the paper that the steps consume and produce;

paths, which represent the established communication links between the various operating entities.

The language makes heavy use of two dimensional programming techniques and is intended to be used with a sophisticated display device. The language is also heavily biased toward interactive applications, in recognition of the fact that many business activities can only be partially automated. This aspect of the language, however, will not be discussed.

BDL programs are developed in a structured way. Programming begins by drawing on a screen labelled boxes representing the principle operating entities within the application area to be automated. These boxes are connected by paths indicating the lines of communication among them. These paths are labelled with the kinds of documents that flow along them. The semantics associated with this part of BDL is that the step from which the path emanates is expected to produce a set (perhaps a singleton set) of documents at some instance of time. The steps receiving this set of documents will begin execution as soon as a set is available along each of their input paths. Thus at this level the language is a data flow language with steps producing all of their outputs simultaneously and executing whenever their input is ready. The only constraints on the order of execution implied are those implied by the data flow constraints.

An example of this aspect of the BDL program is shown in figure 3. This example indicates an application consisting of three steps, namely, Billing, Inventory Control, and Sales Analysis. It involves several sorts of documents: Orders, Order Acknowledgements, Back Orders, Item Summaries, Invoices, and Salesman Summaries. Except for Orders, which originate outside of this step, all of these documents originate in the Billing department and are used to trigger work by the Inventory Control department and the Sales Analysis department, or, in the case of Order Acknowledgements and Invoices, to be sent outside of the system. It should be emphasized that a programmer writing in BDL will, via a display device, enter precisely the sort of diagram shown in figure 3. BDL is not a linear language. This information is part of the BDL program; it is <u>not</u> accompanying documentation.

The application expert then proceeds to specify in more detail how each of the steps executes. As figure 4 illustrates, this can be accomplished in part simply by elaborating the data flow to a further level of detail. In figure 4 the Billing step has been broken down to indicate that it contains three sub-steps: the step which produces the Order Acknowledgement, the step which produces the Invoice and a special step called an accumulator. The accumulator step, like a number of special purpose steps which have been introduced into BDL, is a device which is useful in data flow programming and which, it is hoped, will reduce the programming burden. The accumulator step illustrated here takes in singleton sets of Orders until a signal is received. At that point it outputs the accumulated group of Orders.

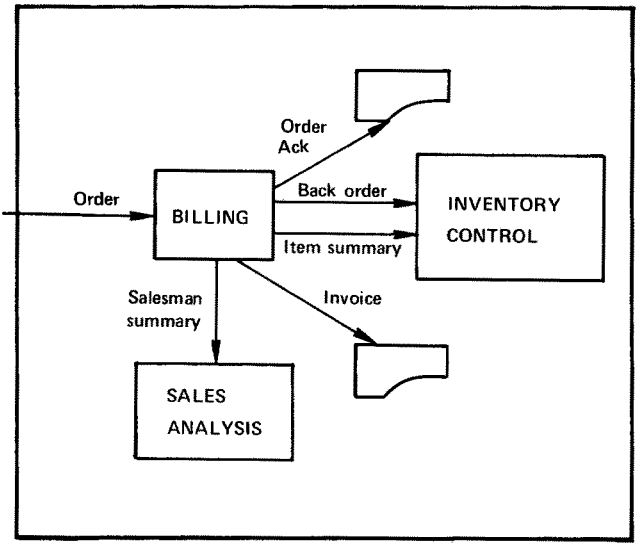

FIGURE 3. AN INITIAL BDL PROGRAM

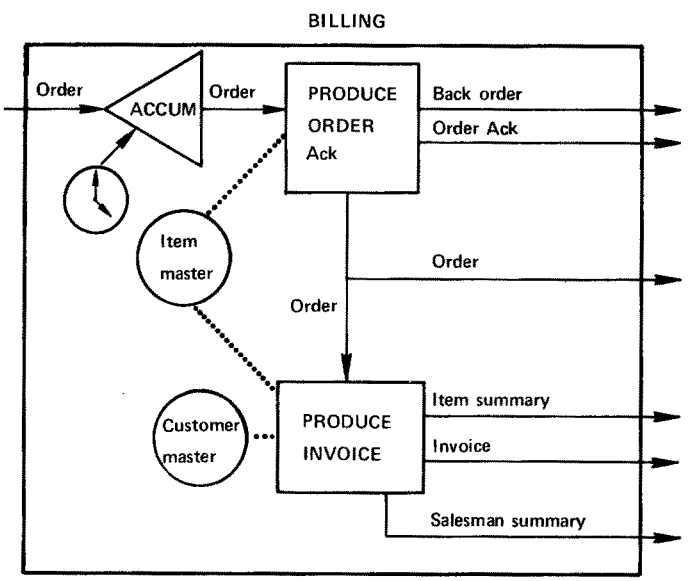

FIGURE 4. FURTHER SPECIFICATION OF THE BILLING STEP

Another facit of BDL illustrated in figure 4 is the role of permanent storage, called files. In figure 4 two files, the Item Master and Customer Master, are shown. The Item Master can be accessed by both the Produce Order Acknowledgement step and the Produce Invoice step. The Customer Master file is accessed only by the Produce Invoice step. The perception of files encouraged in BDL is similar to the one which in a manual operation is associated with a folder in a file drawer: that is, it is a collection of documents all of the same sort which are accessed in total by a step when it executes. Thus from the point of a step, it is as if the step simply had another input path.

The programmer can continue with this process of elaboration as long as he finds it productive. At some point, however, the problem cannot reasonably be decomposed further in terms of data flow. The programmer must indicate the details of the relation between the group of input documents and the group of output documents. At this point the BDL programmer makes use of another component of BDL, a component specifically designed to show how the transformation from input to output documents is accomplished.

To indicate the details of this transformation the programmer utilizes another portion of BDL called the transformation component. The transformation component is a highly structured language with a tabular format. Figure 5 illustrates such a piece of code. The first column lists, in a structure-like format, the fields on the output document to be computed. A field may have sub-fields and a given field may be repeated indefinitely often. The latter possibility is indicated by appending "'s" to the field name in the listing. The second column listed is causality/derivation. For individual (scalar) fields it indicates the computation by which the field value is derived. For indefinitely repeating groups of fields it indicates how to compute the cardinality of this group for a particular document and, in effect, selects the set of items that will be used in computing the subfields. In writing the causality/derivation, field names may be introduced which are not names of fields from the first column. Each such name is listed in the third column. The definition of the name is given in the fourth column.

Figure 5 is the BDL program equivalent to the PL/I program of figure 1. That is, it is the program to calculate invoices assuming that one invoice is to be produced for each incoming order. Several aspects of the transformational component are illustrated by this example. First, the program centers around the computation of the output. This is in contrast to the PL/I program, which was driven by consideration of the input. Second, the form of the transformation component makes it very difficult to reuse computations. This is illustrated by considering the computation for Name

Group/Field	Causality/Derivation	Name	Definition
1 Invoice's	ONE PER Order	Order's	INPUT
2 Name	Customer name	Customer name Assoc. Customer master Customer number Customer master's	IN Assoc. Customer master Customer master WHERE Customer number = Cust# IN Customer master INPUT
2 Address	Customer address	Customer address Assoc. Customer master Customer number Customer master's	IN Assoc. customer master Customer master WHERE Customer number = Cust# IN Customer master INPUT
2 Cust#	Incoming Cust#	Incoming Cust# Order	IN Order CAUSE OF Invoice
2 Item's	ONE PER Incoming Item	Incoming Item's Order	IN Order CAUSE OF Invoice
3 Item#	Incoming Item#	Incoming Item# Incoming Item	IN Incoming Item CAUSE OF Item
3 Quantity	Incoming Quantity	Incoming Quantity Incoming Item	IN Incoming Item CAUSE OF Item
3 Price	Item price	Item price Assoc. Item master Item number Item master's	IN Assoc. Item master Item master WHERE Item number = Item# IN Item master INPUT
3 Extended price	Quantity - Price		
2 Gross	SUM(Extended price's)		
2 Tax	0.05 x STATE(Address) = SUM(Quantity's x "New York" Price's) 0 OTHERWISE		
2 Amount	Gross + Tax		

FIGURE 5. BDL PROGRAM TO PRODUCE INVOICES (ONE PER ORDER) FROM ORDERS

and Address in figure 5. These calculations both require access to the Customer Master file. In most programming languages, assuming the file resides in secondary memory, the program would be written in such a way that the appropriate record was accessed once and the two fields extracted. In BDL this must be indicated by two separate computations. This, of course, permits us to change the definition of how one value is computed without requiring that the second definition be changed as well. This is one way in which BDL supports the notion of locality of definition.

Figure 6, which corresponds to the program in figure 2, illustrates other characteristics of the transformational component. If a program is to be written in terms of the output rather than the input, then it is necessary to indicate what causes a given element in the output set to be produced.

The causality for invoices is given as "ONE PER Like Orders". Like Orders are defined to be "Orders WITH COMMON Cust#". The effect of this set of definitions is to create a partition of the input set, where each element in a partition are the Orders with a common customer number. One Invoice is produced for each element in the partition. Similarily, items are produced "ONE PER Like Incoming Item's" where Like Incoming Item's are defined to be "Incoming Item's WITH COMMON Item#" and where Incoming Items are defined to be "ALL IN Like Order's". This piece of code first causes a set to be formed of all Items on all Like Orders (that is all Orders with a given customer number). This set of items is then partitioned according to common item number and an item on the Invoice is produced for each element in the partition.

Observe that the two programs in figures 5 and 6 vary only in those parts of the program which are concerned with calculating Invoices and Items. All other computations remain the same. This is another aspect of which BDL supports the notion of locality of reference.

Group/Field	Causality/Derivation	Name	Definition
1 Invoice's	ONE PER Like Order's	Like Order's Order's	Order's WITH COMMON Cust# INPUT
2 Name	Customer name	Customer name Assoc. Customer master Customer number Customer master Customer master's	IN Assoc. Customer master Customer master WHERE Customer number = Cust# IN Customer master INPUT
2 Address	Customer address	Customer address Assoc. Customer master Customer number Customer master Customer master's	IN Assoc. customer master Customer master WHERE Customer number = Cust# IN Customer master INPUT
2 Cust#	Incoming Cust#	Incoming Cust# Order	IN Order CAUSE OF Invoice
2 Item's	ONE PER Like incoming Item's	Like incoming Item's Incoming Item's Like Order's	Incoming Item's WITH COMMON Item# ALL IN Like Order's CAUSE OF Invoice
3 Item#	Incoming Item#	Incoming Item# Like incoming Item's	COMMON IN Like incoming Item CAUSE OF Item
3 Quantity	SUM(Incoming Quantity's)	Incoming Quantity Like incoming Item's	IN Like incoming Item CAUSE OF Item
3 Price	Item price	Item price Assoc. Item master Item number Item master's	IN Assoc. Item master Item master WHERE Item number = Item# IN Item master INPUT
3 Extended price	Quantity - Price		
2 Gross	SUM(Extended price's)		
2 Tax	0.05 x SUM(Quantity's x Price's) STATE(Address) = "New York" 0 OTHERWISE		
2 Amount	Gross + Tax		

FIGURE 6. BDL PROGRAM TO PRODUCE INVOICES FROM ORDERS (GROUPED BY CUSTOMER)

There are other components of BDL which are used to complete the program definition. One of these components has to do with the definition of forms, i.e., empty documents. In this component are isolated all of the data typing, data formatting and field sizing aspects of a program which, in more conventional languages, are sprinkled throughout the program. These definitions are accomplished by displaying or creating a form on a screen and systematically filling it out in various ways: indicating either the name, the field size, the formats, or the data types of the objects which it is to contain. This not only isolates these aspects of programming but also is a specific context in which an application expert can deal with these matters. In passing, it is interesting to note that BDL contains several unconventional data types having to do with money, dates, addresses, etc. It also has rather strict rules about the ways in which different data types can be combined and what the resulting data type is. These rules are used to do extensive type checking on the computations used in the transformational component.

Finally, there are ways for indicating that a transformation step involves human interaction. For all steps associated with an I/O device there are ways of indicating this as well.

In summary, then, BDL is a highly structured language in which the various components of programs in an application area are isolated and dealt with separately. This maximizes maximal possibilities of changing one aspect of a program without concern for other aspects. This in turn gives us the manipulability and composability that we seek.

V. Other Associated Work

BDL has been designed to make it possible for an application expert to write a large set of program fragments which, taken as a whole, can be used to generate a large variety tailored application programs. Two issues remain to be discussed. The first concerns the method by which information is extracted from the user concerning his application and how that information is used to generate the programs required for his application. We are currently working on two quite different techniques for accomplishing this purpose.

In the first approach we are attempting to judge the feasibility of using a predetermined set of questions, which are chosen by the application programmer who writes the program fragments. The application programmer is also given straightforward methods for showing the composition of program fragments corresponding to the various alternative answers to the question. The system is

also being designed so that the application programmer can introduce his question set without regard to the order in which the eventual end user will answer the questions. Although it is feasible to build such a system it is less clear whether it is possible for the application expert to generate such a question set and the associated actions on the program fragments and, further, whether such a system will satisfy the user's needs. We are currently building a prototype system of this sort and plan to run extensive human factor studies to determine how easy it is both for the application expert and for the end user.

A second approach to the question of eliciting information from the end user is to try to build a system which engages in a more informal dialogue and which, as a result of an extended conversation with the user, determines his application needs and arranges for the appropriate program to be generated. Here the problem is not so much the specification of an appropriate interface as the development of a methodology to support it. As a consequence we are investigating the issue of how to model programs and how to model the business world. We are also formulating methodologies for inferring the information required for generating appropriate programs. Another important issue is how to exploit these models to direct the conversation with the user in an appropriate direction so that the necessary information can be gathered. We must however, be prepared to accept unexpected information from the user and make use of it as best we can. Obviously, we must be able to uncover and resolve apparent inconsistencies and misunderstandings on the part of the user. Clearly, this is a very difficult task, and one in which we do not expect immediate results.

Another problem with which we must deal is the translation of the BDL programs to executable code. BDL as a language forces considerable redundancy in the specification of a system. This redundancy is an especially severe problem because many of the operators in BDL are aggregate operators. Because of its forms orientation, this redundancy also appears in the data structures used in a given program. As a consequence straightforward implementation of BDL, either as an interpreter or as a simple compiler, will result in unacceptable execution time. Hence we are looking at program analysis and optimization techniques which will transform a program written in BDL which takes maximum advantage of the application expert's skill into a program that makes better utilization of the machine resources. This analysis must take place on the whole complex of BDL modules. It cannot be limited to a single transformational step. Considerable emphasis is being placed on the collection of global information relating the various parts of the BDL program, and on transformations involving the aggregate operators and data structures.

VI. Acknowledgements

The work briefly outlined above corresponds only to one facit of the work in Automatic Programming at Yorktown Heights. Other points of view are also being vigorously pursued. These will be reported on at a later date. The work outlined includes contributions from a large number of members of the Automatic Programming group. Special acknowledgement should be made of the contributions of Gerry Howe, Irving Wladawsky, Vincent Kruskal and Mike Hammer (now at MIT) in the definition of BDL. Irving Wladawsky, Martin Mikelsons, Peter Sheridan and George Heidorn are responsible for the work on the Information Acquisition System. Fran Allen, Dave Lomet, and Bill Harrison are contributing the design of the analysis and optimization techniques.

References

[1] Balzer, Robert, Automatic Programming, Technical Memo, Information Sciences Institute, University of Southern California, September, 1972.

[2] Martin, William, et. al., Automatic Programming Internal Memos, 1972, 1973.

[3] Automatic Programming Workshop, M.I.T., January, 1973.

[4] Hershey, E. A., et. al., PSL/II Language Specifications, Version 1.0 ISDOS Working Paper No. 68, University of Michigan, Dept. of Industrial and Operations Engineering, Ann Arbor, Michigan (Feb. 1973).

[5] Hammer, M. M., Howe, W. G., Wladawsky, I., An Interactive Business Definition System, RC 4680, IBM T. J. Watson Research Center, Yorktown Heights, New York, January, 1974.