LABORATORY FOR
COMPUTER SCIENCE
(formerly Project MAC)

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-72

PROTOSYSTEM I:
AN AUTOMATIC PROGRAMMING SYSTEM PROTOTYPE

GREGORY R. RUTH

JULY 1976

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

MIT/LCS/TM-72

Protosystem I:

An Automatic Programming System Prototype

Gregory R. Ruth

July 1976

This work has been supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract #N00014-75-C-0661

Massachusetts Institute of Technology

Laboratory for Computer Science

(formerly Project MAC)

CAMBRIDGE MASSACHUSETTS 02139

Protosystem I: An Automatic Programming System Prototype

ABSTRACT

A model of the data processing system writing process is given in terms of development stages. These stages correspond to the progression in the implementation and design process from the highest level of abstraction (English system specifications) to the lowest level (machine code). The issues and goals (including optimization of the product data processing systems) involved in automating these stages are discussed and strategies and methodologies used for doing so are developed.

Protosystem I, an automatic programming system prototype, is described. The completed (and working) part automates three of the five stages identified in the proposed model of the system writing process. The basic theory, methods and structure of this part of the automatic programming systems are presented.

Protosystem I: An Automatic Programming System Prototype

Programming is the activity of going from a task specification to code capable of performing the task on some actual computing system. This is essentially a problem solving process. But over the years people have come to understand certain functions of the programming process well enough to automate them--that is to replace those functions by programs. The most notable results were assemblers, compilers, and operating systems. The gains realized from this automation were reduced operating errors, increased complexity of systems which could be completed and more efficient use of resources (time, people, machines) in the design-implement-evaluate cycle.

Our knowledge and understanding of programming is once again reaching a level where a significant advance in automation is both necessary and possible. In fact, I believe that the entire programming process can now be effectively automated. That is, a system can be developed that will engage the user in English language discourse about a desired task and will produce, as a result of the interaction with the user, a satisfactory program. To demonstrate feasibility and gain insight into the issues and technology involved in creating such a system, a prototype automatic programming system (Protosystem I) for generating business data processing systems is currently being developed at MIT.

A Model of the Program Writing Process

The data processing system writing process may be conceived as a sequence of phases leading from the conception of a system to its implementation as executable machine code. A useful and plausible model for this sequence of phases is:

Phase 1: Problem Definition (English → OWL)

The system specification is expressed in domain dependent terms in English that is understandable by the program developers.

Phase 2: General System Analysis and Design (OWL → SSL)

The problem is reformulated in standard data processing terms and expressed as an instance of a known solvable problem class (in our case a subset of the class of all batch oriented dps's). Domain dependent policy and procedures are worked out in detail at this stage.

Phase 3: System Implementation (SSL → CDSL)

The system--the actual procedural steps and data representations and organizations--is constructed by intelligent selection from and adaptation of a number of standard implementations possibilities.

Phase 4: Code Generation (CDSL → PL/I & JCL)

The design specifications are implemented in a high-level language (e.g. PL/I, COBOL) in a fairly straightforward, but not totally mechanical, way.

Phase 5: Compilation and Loading (PL/I & JCL → machine executable form)

A form is produced that that can be "understood" and executed by the target computer.

These phases progress from a general notion of *what* is to be done by the desired system toward a detailed specification of *how* it can be accomplished. They also represent the classes of design and implementation problems involved in program writing, progressing from the most global and general considerations toward the most local and detailed issues.

Protosystem I seeks to automate the program writing process by automating and tying together the phases described in the model given above. That is, Protosystem I is

designed in such a way that there are explicit parts or stages corresponding to each of the model phases. Each such stage embodies the knowledge and expertise of the human agent(s) for the corresponding phase, so that, given the same or similar input, it can intelligently produce comparable corresponding results. Drawing on experience gained in recent artificial intelligence and knowledge based systems research, we have chosen to represent the knowledge in each stage in the form of procedures as opposed to the approach used, for example, in table driven compilers.

The products of each stage should not be so rigidly deterministic so that the courses of action in further stages are narrowly prescribed. They must be sufficiently general and malleable so that further stages can have the maximum freedom in making their design contributions in the most effective and efficient ways. Consequently, we have chosen in Protosystem I to make the product of each stage a descriptive representation of the dps in terms of concepts and considerations appropriate for the next stage of development. Such a description provides a medium in which the next stage can manipulate relevant concepts and analyze the dps for relevant properties, so that it can perform its design job more in the manner of a problem solver than in that of an automaton. In this way the programming process is conceived as the development of a succession of ever more precise system descriptions until, ultimately, a level is reached where every detail has been decided and the result is an executable computer program.

Put another way, we have borrowed from structured programming the notion of program development by successive refinement, but we have approached this by extending the levels of language--machine, assembly, and compiler--to include the very high level language, SSL, and an English-like language, OWL. We recognize that to succeed with

this method one must make appropriate abstractions so that the more abstract statements of the problem lend themselves to further refinement. While the wide use of business data processing systems in the past two decades has not led to the definition of standard modules which would handle any situation, it has led to a better understanding of the useful abstractions in that field. We believe that these abstractions can be grouped naturally according to the phases of the dps development process.

Efficiency Enhancement in System Development

To produce a credable and practical result an automatic programming system must perform a reasonable degree of optimization. Current formal optimization methods pertain mainly to the compilation level, principally because this is the only phase of the program writing process that has been automated. When the entire program development process is automated, new, additional types of optimization will have to be included. The combination (for the sake of I/O efficiency) of computations accessing the same data set is an example. At compilation time the decision to include these computations in the same job step or not to do so has already been made. If they are not in the same job step (and hence compiled separately) the opportunity for optimization has already been lost. Even if they are in the same job step, the compiler can derive little, if any, information about their I/O characteristics. Therefore, it has no basis on which to evaluate the relative efficiency of possible combinations and is incapable of making an intelligent optimization decision.

This type of optimization problem in not unique. It should be easy for the reader to think of many other examples where it is impossible to perform adequate

optimization of the type necessary if we wait until Phase 5 to do it. The information needed to make good design decisions of a more global nature is just not available at that stage. That is, the (so-called "high level") compiler language is too low level to allow the system specification to be expressed in a form where such optimization issues will be apparent.

The various possible types of optimizations fall quite naturally into categories that correspond to the program writing levels in our model. For instance, the combination of computations as in the above example is something that should be considered during Phase 3 (system implementation) where the data and computational interrelationships among conceptual processing units are most evident. Problems involving machine language inefficiencies should be handled in a later phase.

An attempt to apply an optimizing process at a higher stage than that to which it pertains would require an overspecified system description at that level; that is, a description containing details extraneous to the purpose of that stage. Trying to apply a transformation at a later stage than that to which it naturally corresponds would effectively require "unprogramming" (translation from a lower level description to a higher level one). This would be a difficult, if not impossible, task. It would also require the lower phase to contain knowledge which belongs at a higher level. Optimizations are most effectively performed at their corresponding level of translation, where exactly the sort of information and visibility needed is present. Since all levels of program writing are included in our automatic programming system, there is no need for overspecification or unprogramming.

At each stage in an automatic programming system intelligent, efficiency enhancing design and implementation require: knowledge of possible ways of implementing

each requirement determined and described by the previous stage, methods for evaluating alternate designs so that the best choice can be made, and information on which such decisions can be based. The first two of these embody the expertise of the human agents for the the corresponding phase of each stage and are built into the programs for that stage.

The information used to make decisions is specific to the particular dps construction project. It consists of three parts: (1) the design decisions that have been made so far for the dps part under consideration and for all parts relevant to the further development of that part, (2) the consequences that a decision will have on further development, and (3) the environment in which that part and related parts will operate. The first of these is provided by the description output of the previous stage, which is input to the present stage. It also includes the current partially determined description of the dps being produced by the present stage. This contains the decisions that have been made so far in this stage.

The second type of information is of two kinds: (a) knowledge of the effects of a decision on others to be made in the same stage and (b) knowledge of its effects on decisions to be made in later stages. Information of type (a) is provided by maintaining a global awareness within each stage of its aggregate design contribution. Type (b) information is commonly provided by feedback in the non-automated program development process. Consider the problem of avoiding machine language inefficiencies. A number of such inefficiencies can be eliminated simply by finding better sequences of instructions to implement the constructs of the next higher level language; but others can only be prevented by using only algorithms (which are determined at a higher level) that

require constructs that can be efficiently implemented. In the latter case one could imagine a strong interaction between levels, for in choosing an algorithm it would be necessary to determine whether the constructs it requires could be implemented efficiently. We believe that in the semi-repetitive design of business data processing systems this strong interaction is not required. We eliminate most difficulties by removing from the design any algorithm which has not been part of an efficient implementation.

We can do this because our goal is not to be the first to create radically new systems, but to implement standard systems quickly, cheaply, and accurately. The semi-repetitive nature of the design of dps software also makes it possible to get ball-park estimates of cost without detailed coding of key sections. One could cite, for example, estimating systems like SCERT which have been in commercial use for some time.

As stated above, we take the view (also held by some structured programming advocates) that one design phase should be completed before the next is begun. Because we are automating the design process, we can make many passes through the whole process in the time formerly required for a single pass. Feedback from several passes, including evaluation by the using organization, we hold to be critical. Feedback among phases as they run greatly complicates the design process. We feel that it will not yield a corresponding improvement in results. It opens the door to the design of incredibly complex "heterarchical" systems where control of the program development process would dance unpredictably among the various stages in a complicated way. To avoid the complexities of a heterarchically structured system, we have outlawed feedback. Instead, each stage has a gross model (often implicit) of the stage following it. In this way it can see the basic ramifications of its design decisions in the next stage by effectively interrogating its own

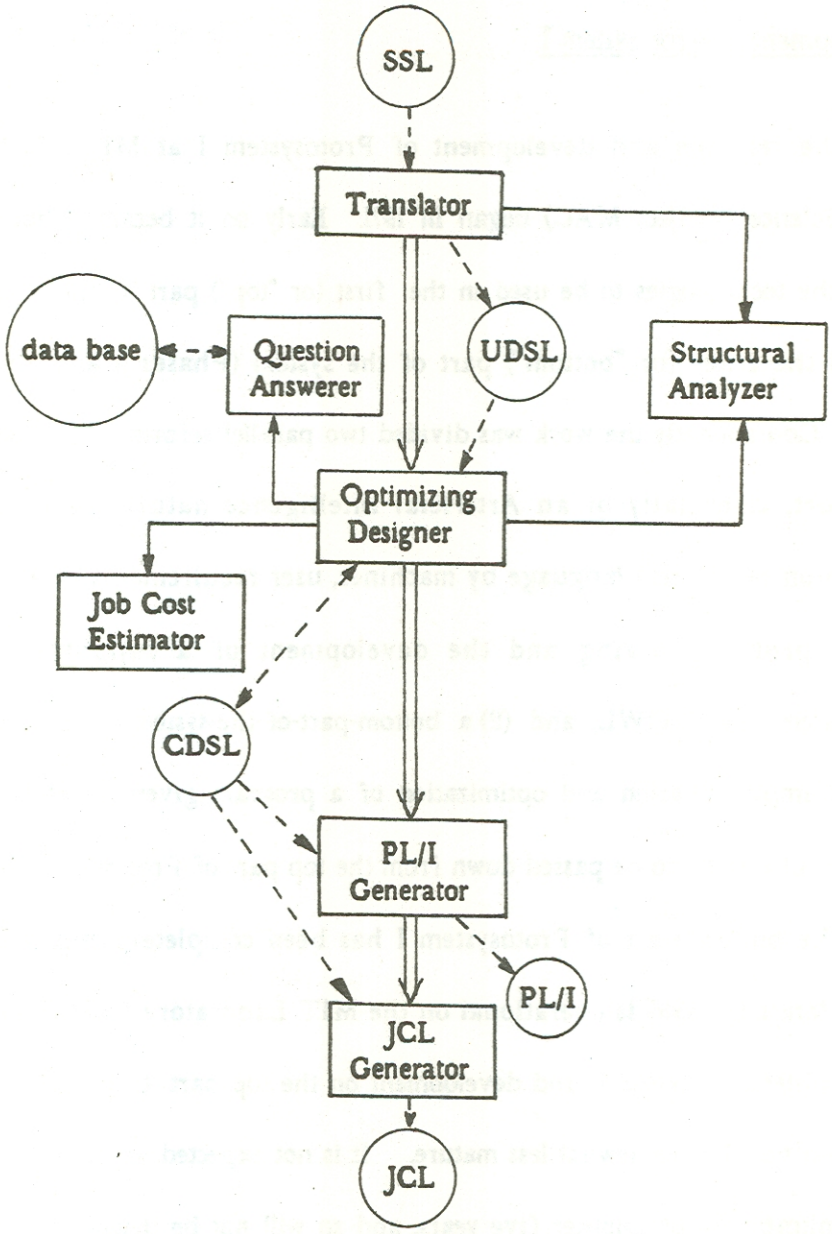
model of that stage, rather than having to rely on that stage to feed information back.

The third type of information needed in the design and implementation process has to do with the context in which the dps will operate, namely: (a) the machine/operating-system configuration on which the ultimate dps code will be executed and (b) the characteristics of the data it will receive and produce. Because machine/operating-system configurations are standard and relatively few in number, type (a) information is encoded directly into the automatic programming system in the form of separable, interchangeable modules; the automatic programming system is thus specialized to a particular configuration by selecting the appropriate module (e.g. the OS/360 module). Further installation dependent information (e.g. the number and type of secondary storage devices) is supplied directly by the user. Information of type (b), concerning types and quantities of, and interrelationships among, data processed by the dps, is too broad and varied in nature to be entirely supplied by, or derived from, an initial user statement of any reasonable length. The inclusion of all facts that might possibly become relevant to design decisions would require much effort. Much information is difficult to obtain and most would never be used. Therefore, it is best to require special information from the user only when it becomes important in the course of the development process. To do this the automatic programming system must be able to ask the user specific questions as additional information becomes necessary.

The Development of Protosystem I

The research and development of Protosystem I at MIT's Laboratory for Computer Science (Project MAC) began in 1971. Early on it became apparent that the natures of the technologies to be used in the first (or "top") part of the system (Phases 1 and 2) and the latter (or "bottom") part of the system (Phases 3 and 4) were clearly different. Consequently the work was divided two parallel efforts: (1) a top-part-of-the-system effort, essentially of an Artificial Intelligence nature, involved with the comprehension of natural language by machines, user requirements acquisition, model formation, problem solving and the development of a supporting high-level language/system called OWL, and (2) a bottom-part-of-the-system-effort addressing the problems of implementation and optimization of a program given an abstract relational specification (ultimately to be passed down from the top part of Protosystem I) of what it is to do. The bottom part of Protosystem I has been completely implemented in the MACLISP language and is operational on the MIT Laboratory for Computer Science PDP-10 computers. Research and development on the top part, being considerably more ambitious and novel, is somewhat less mature. It is not expected to cross the threshold of practical applicability for another five years, and so will not be discussed further in this article.

A structural diagram, indicating the major modules, flows of control and flows of data in the bottom part of Protosystem I is shown in Fig. 1. The following sections give an explanation of the working of this part of the automatic programming system.



Data flow - - - -
Calls ————
Transfer of control ═══

Figure 1

Protosystem I: Structure of the Bottom Part

The Protosystem I Data Processing System Model and the System Specification Language

Protosystem I handles a restricted but significant subset of all data processing applications: I/O intensive batch oriented systems. Such systems involve a sequence of runs or job steps that are to be performed at specified times. They are assumed to involve significant I/O activity due to repetitive processing of keyed records from large files of data. Systems such as inventory control, payroll, and employee or student record keeping systems are of this type.

A simple example of such a dps is a software system to perform the inventory and warehousing activities in the following case:

> The A & T Supermarket chain consists of 500 stores served by a centrally located warehouse. There are 4000 items, supplied by the warehouse, that these stores can carry.
>
> Every day the warehouse receives shipments from suppliers and updates its inventory level records accordingly.
>
> It also receives orders from the stores for various quantities of items. If for a particular item there is sufficient stock to fill all of the orders for that item, the warehouse simply fills the orders as made; but if there is insufficient stock it ships partial orders proportional to fraction of the total quantity ordered that is on hand.
>
> Inventory records are adjusted to reflect the decrease in levels.
>
> Finally, a daily check is made on the inventory levels of all items. If the level of an item is lower than 100 the warehouse orders 1000 more units of that item from the appropriate supplier.

In order for the bottom part of Protosystem I to implement such a data processing system application the basic aggregate data entities and their interrelationships must be determined. This determination can be made from the English task description by a consultant or by a sophisticated, natural language comprehending software system (e.g. the top-part of Protosystem I) that has embedded in it his knowledge and experience about

business systems and data processing.

Consider the inventory updating activity of the second paragraph. There are three aggregate data entities involved: (1) the set of quantities received from suppliers, (2) the set of closing inventory levels for the previous day, and (3) the set of the updated inventory levels to be used for filling store orders. Such collections of similar data that are to processed in a similar way are termed *data sets*. In the domain of Protosystem I a data set is assumed to consist of fixed format *records* (e.g. one for the level of each inventory item). Associated with each record is a *data ttem* (e.g. the level of an inventory item) and *keys*. The key values of a record uniquely distinguish it (e.g. the inventory data set can be keyed by item since there is only one level [record] per item) and so can be used to select it. Thus, a data set is essentially the same as a Codd relation and its keys are what Codd calls primary keys.

Let us call the three data sets described in the last paragraph SHIPMENTS-RECEIVED, FINAL-INVENTORY and BEGINNING-INVENTORY. The relationship between the BEGINNING-INVENTORY data set and the SHIPMENTS-RECEIVED and FINAL-INVENTORY data sets may be described as:

For every item:

> the beginning inventory level of that item
> (i.e. the value of the data item for the record in BEGINNING-INVENTORY for that item)
>
> is the closing inventory level of that item from the previous day
> (i.e. the value of the data item in the record of FINAL-INVENTORY for the same item)

plus the quantity of that item received
(i.e. the value of the data item in the record of SHIPMENTS-RECEIVED
for the item in question),

if any.

This relationship is expressed more succinctly in SSL (the System Specification Language):

```
BEGINNING-INVENTORY IS FINAL-INVENTORY(1 DAY AGO) + SHIPMENTS-RECEIVED
```

Implicit in this statement is that the addition operation is performed for each item and that if one of the operands is missing (e.g. if no chicken noodle soup was received today) it is treated as having a zero value. The repetitive application of an operation to the members of a data set or sets such as this is termed a *computation.* The order of applications of the operation to the records of its input data sets by a computation is assumed to be unimportant to the user; in fact, he may think of them as being performed in parallel. However, every computation does, in fact, process its inputs serially, according to a particular ordering (chosen by Protosystem I) on their keys. Computations typically *match* records from different data sets by their keys (as above) and operate on the matching records to produce a corresponding output record. A computation may also *group* the members of a data set by common keys and operate on each group to produce a single corresponding output. Returning to our example, note that item orders can come from different sources (stores), so that both the item and the source of an order are needed (as keys) to distinguish it. To form the total of all orders for each item, a computation must group the orders by item and sum over the order amounts in each group. In SSL this would be expressed as:

```
TOTAL-ORDERS FOR EACH ITEM IS THE SUM OF THE QUANTITY-ORDERED-BY-STORE
```

Fig. 2 shows the structure of the A & T inventory and warehousing data processing system in terms of computations (boxed) and data sets (unboxed). The complete SSL description of A & T dps is given in Fig. 3. Note that in addition to the relational statements a list of data sets must be included to indicate the keys by which they are accessed.

The Translator and the Data Set Language

It is characteristic of the data processing systems which Protosystem I proposes to treat that the calculations themselves are easily dealt with and that it is the structuring and manipulation of the masses of data involved that occupies by far the greater part of the Stage 3 implementation activity. Additionally, the moving and storage of aggregate data entities must be determined before the operations on their members can be considered. Consequently, the development process at Stage 3 is data set oriented. Therefore, to facilitate the design process the SSL dps description is first analyzed from this point of view and re-expressed in a more appropriate medium, DSL (the Data Set Language). This reformulation is performed by the Translator module.

The determination of dps characteristics that can aid in the development of the dps design is made with the aid of the Structural Analyzer and included in the Translator's output description. This output is called the UDSL (Unconstrained Data Set Language) description, because most design details remain unbound (undecided) in it. As such it forms the skeleton of the dps description ultimately to be produced by Stage 3.

One useful piece of information determined by the Structural Analyzer is the set

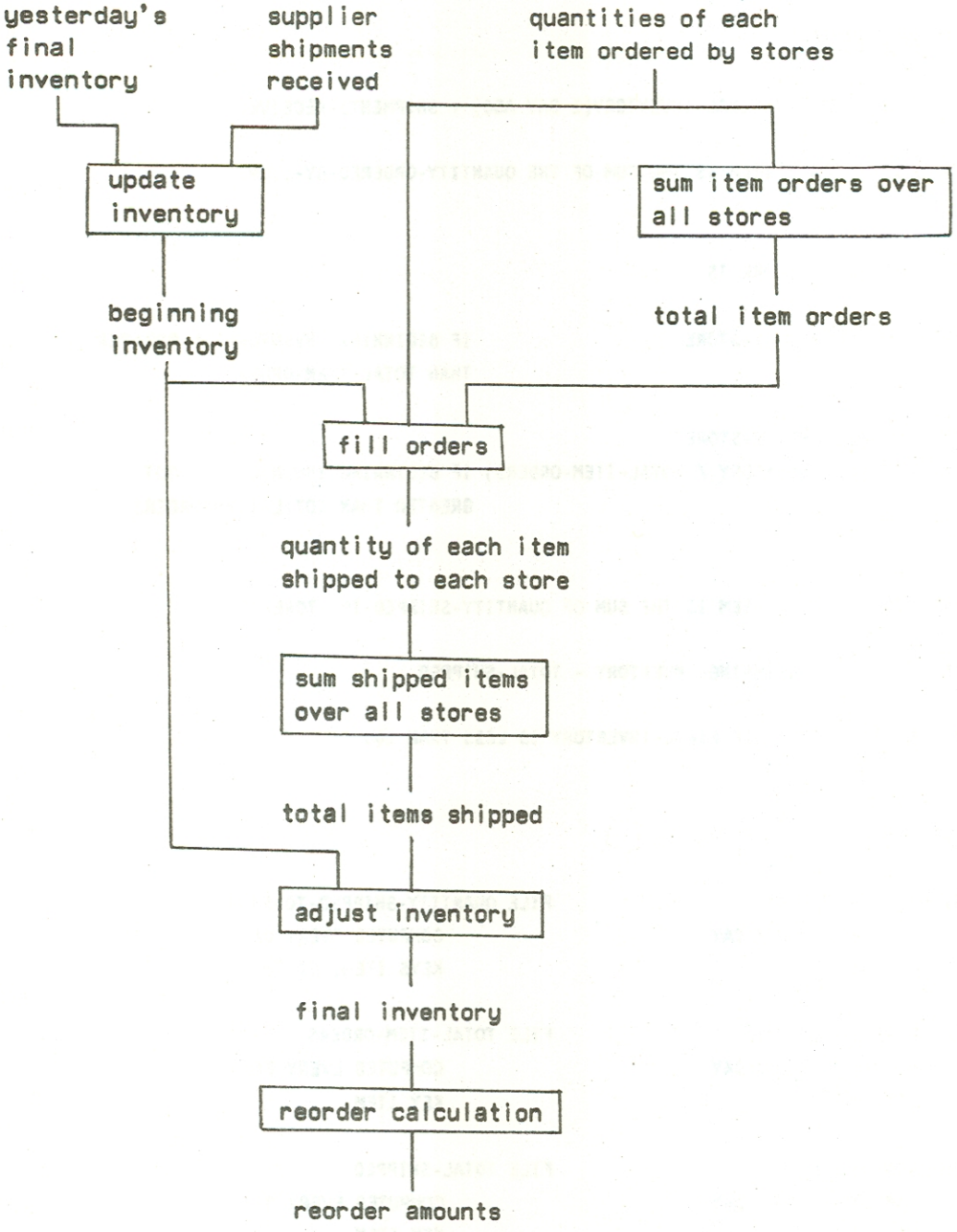


Figure 2
A & T Inventory and Warehousing System

```
COMPUTATION DIVISION

BEGINNING-INVENTORY IS FINAL-INVENTORY(1 DAY AGO) + SHIPMENTS-RECEIVED

TOTAL-ORDERS FOR EACH ITEM IS THE SUM OF THE QUANTITY-ORDERED-BY-STORE


QUANTITY-SHIPPED-TO-STORE IS

     QUANTITY-ORDERED-BY-STORE                    IF BEGINNING INVENTORY IS GREATER
                                                  THAN TOTAL-ITEM-ORDERS

     QUANTITY-ORDERED-BY-STORE
     * (BEGINNING-INVENTORY / TOTAL-ITEM-ORDERS) IF BEGINNING INVENTORY IS NOT
                                                  GREATER THAN TOTAL-ITEM-ORDERS


TOTAL-SHIPPED FOR EACH ITEM IS THE SUM OF QUANTITY-SHIPPED-TO-STORE

FINAL-INVENTORY IS BEGINNING-INVENTORY - TOTAL-SHIPPED

REORDER-AMOUNTS IS 1000 IF FINAL-INVENTORY IS LESS THAN 100


DATA DIVISION

FILE SHIPMENTS-RECEIVED                  FILE QUANTITY-SHIPPED-TO-STORE
          COMPUTED EVERY DAY                       COMPUTED EVERY DAY
          KEY ITEM                                 KEYS ITEM, STORE

FILE BEGINNING-INVENTORY                 FILE TOTAL-ITEM-ORDERS
          COMPUTED EVERY DAY                       COMPUTED EVERY DAY
          KEY ITEM                                 KEY ITEM

FILE FINAL-INVENTORY                     FILE TOTAL-SHIPPED
          COMPUTED EVERY DAY                       COMPUTED EVERY DAY
          KEY ITEM                                 KEY ITEM

FILE QUANTITY-ORDERED-BY-STORE           FILE REORDER-AMOUNTS
          COMPUTED EVERY DAY                       COMPUTED EVERY DAY
          KEY ITEM                                 KEY ITEM
```

Figure 3
SSL Relational Description for the A & T Data processing System

of *driving data set* candidates for each computation. A driving data set is an input data set that is guaranteed to have a data item for every tuple of key values for which the computation can produce an output. The computation, then, instead of having to loop over all possible combinations of values for the keys of the inputs, can be driven by the driving data set in that it only has to consider those key value combinations for which the driving data set contains records.

Another type of information the Structural Analyzer determines is directly related to our desire to specify data set organizations and orders and computation accessing methods and orders in such a way as to minimize the cost of operating the dps. Because a dps typically involves the repetitive application of *simple* calculations to large quantities of data we make the first-order approximation that the cost of operation is due entirely to data accessing (reading and writing). Our design, therefore, focuses on minimizing the total number of I/O events.

Accordingly, the Structural Analyzer also determines predicates that are the conditions under which a data item will be generated and under which a data item will be used by a computation. For example, a store will be shipped an item if (it is true that) that store ordered that item and there was sufficient inventory to fill the order; the order allocation step will use the inventory level for a particular item if some store ordered it. These predicates, together with basic information concerning the sizes of data sets in the dps, are used by the Question Answerer to determine the average and maximum sizes of files (proposed by the Optimizing Designer) and the average number of a file's records a computation will access.

The Design Criterion and the Job Cost Estimator

The design criterion for Protosystem I is the minimization of the dollars and cents cost of running the final dps program on the target machine/operating system configuration. Because the dps's are assumed to be I/O intensive, as a first approximation, this can be equated with access minimization. An *access* in this sense is defined as the reading or writing of a single secondary storage block, which corresponds to a single operating system I/O event. In Protosystem I, for a particular data set a *block* consists of a fixed number of records.

With this approximation the relative costs of alternative dps design configurations can often be assessed without knowledge of the particular target configuration. But sometimes actual cost estimates, provided by the Job Cost Estimator, are necessary. This module must thus contain knowledge of the charging scheme and operating characteristics of the target configuration (in our case the OS/360 configuration). Optimization with respect to a different configuration and/or charging scheme would require the substitution of a new appropriately tailored module.

The Question Answerer

The function of the Question Answerer is to supply answers to questions from the Optimizing Designer about the average sizes (in records) of abstract aggregate data entities. Two examples of such data aggregates are a file and the collection of records in a file that are accessed by a particular computation. Each "question" sent to the Question Answerer is in the form of a predicate describing the conditions under which a record will be in the

data aggregate in question. For example, if there are records in FINAL INVENTORY, QUANTITY RECEIVED and BEGINNING INVENTORY for only those items have non-zero quantities, the the predicate

there is a record in FINAL INVENTORY (for a given item)
or
there is a record in QUANTITY RECEIVED (for a given item)

describes an event equivalent to "there is a record in BEGINNING INVENTORY" for a given item. The Question Answerer makes use of the simplifying assumption that all records in an abstract aggregate data entity are equally likely to be present. Thus, if the maximum size of a data aggregate is well defined (e.g. BEGINNING INVENTORY can be no larger than the set of all items carried by the warehouse), its average size can be calculated by multiplying the probability that the event that the typical record in it will be present by its maximum size. If there is no meaningful maximum size (as, for example, with a data set that is the collection of all outstanding purchase orders) the average size of the data aggregate must be determined directly.

The Question Answerer maintains a data base of all of the event probability and size information given by the user. When asked a question it attempts to find the associated size or probability directly. Failing this, it will try to calculate the probability of the event in question happening from those of its sub-events and its knowledge of event independence and correlation within the dps. If the information on hand is insufficient to answer the question, the Question Answerer obtains enough additional information from the user (through a flexible line of questioning) to do so. The new information thus gained is stored in the data base for future reference.

The Optimizing Designer

The Optimizing Designer is the heart of Stage 3; all of the other modules in this stage exist merely to serve it. When the translation from SSL to UDSL has been completed, control passes to the Optimizing Designer. This module is responsible for constructing job steps to implement computations and files to implement data sets. In particular its job is to:

1. design each keyed file--in particular its
    a. contents (information contained)
    b. OS/360 organization (consecutive, index sequential, or regional(2))
    c. storage device
    d. associated sort ordering (by key values)
    e. blocking factor (number of records per block)
2. design each job step of the dps--namely
    a. which computations it includes
    b. its accessing method (sequential, random, core table)
    c. its driving data set(s)
    d. the order (by key values) in which it processes the records of its input data sets
3. determine whether sorts are necessary and where they should be performed
4. determine the sequence of the job steps

The Optimizing Designer performs dynamic analysis (analysis of the operating behavior) on the dps to propose and evaluate alternative design configurations. Occasionally, static

analysis (analysis of system structure and interrelationships) of such tentative configurations is also necessary, and this is obtainedd through calls to the Structural Analyzer. When additional information is needed to make evaluations and decisions the Question Answerer and the Job Cost Estimator are called.

All design decisions are made in an effort to minimize the total number of accesses that must be performed in the execution of the dps. There are three major techniques that the Optimizing Designer uses toward this end:

1. Designing files and job steps in such a way as to take advantage of blocking Accesses can be reduced if files are given blocking factors greater than one and if processing and file organizations are designed in such a way that the records of a each block can be used consecutively.

2. Aggregating data sets If two or more data sets that are accessed by the same computation are combined into one file (see Fig. 4) and processing is arranged so that a single record of the aggregate can be accessed where more than one record from each of the otherwise unaggregated files would have been accessed, accesses can be saved.

3. Aggregating computations When two or more computations access the same data set and the orders in which they process the records of that data set are the same, it may be advantageous to combine them into a single job step. Then each record of the shared data set can be accessed once for all, rather than once for each computation (see Fig. 5).

These access minimizations techniques require that the key order of processing agree in a special way with the organization of the data being processed. This is where

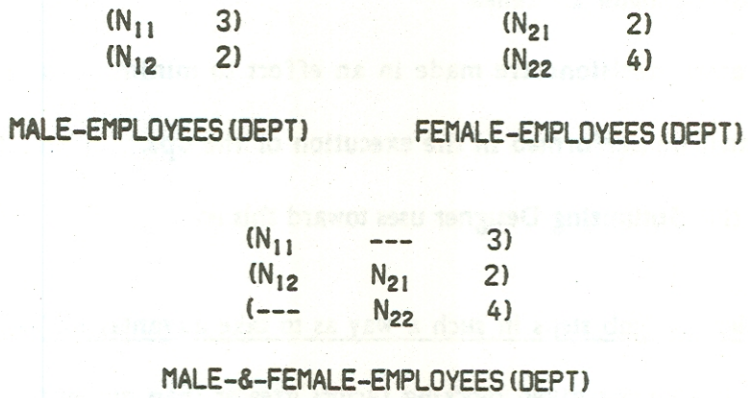


Figure 4: Data Set Aggregation

(The N_{ij} are values of data items and the numbers are values of the key DEPT)

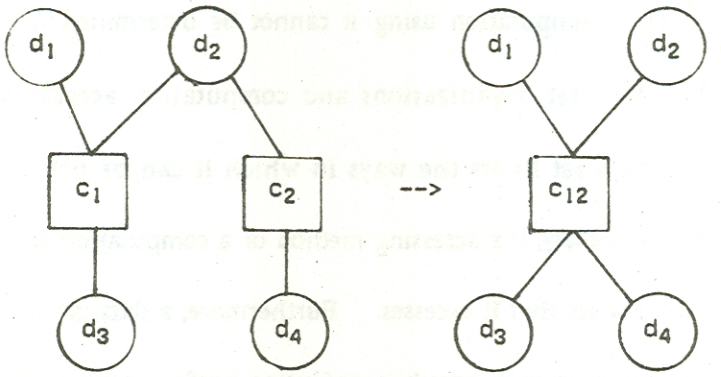


Figure 5.a: Horizontal Aggregation of Computations

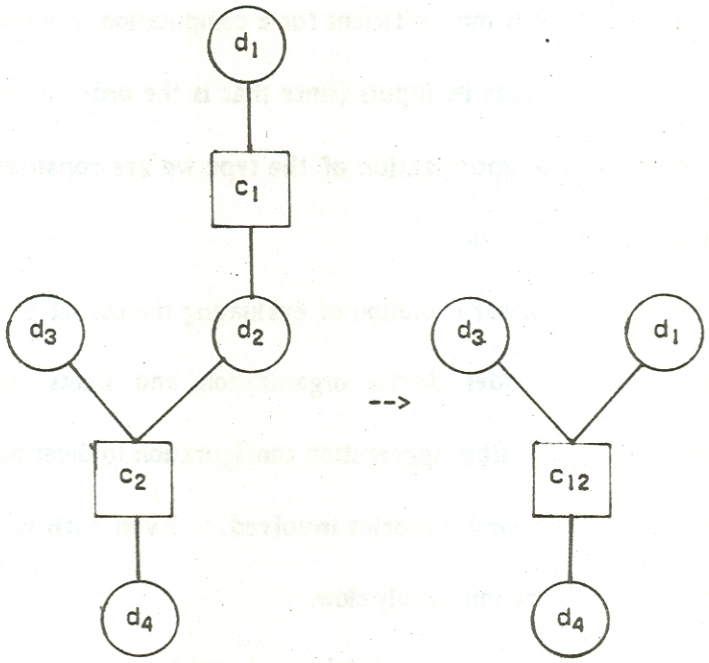


Figure 5.b: Vertical Aggregation of Computations

the fundamental difficulty in optimization lies. A data set's organization and the accessing method of a computation using it cannot be determined independently of each other or of other data set organizations and computation accessing methods. The organization of a data set limits the ways in which it can be practically accessed by a computation, and, conversely, the accessing method of a computation restricts the practicable organizations of a data set that it accesses. Furthermore, a data set is typically accessed by more than one computation with possibly conflicting preferences for its organization; and a computation accesses more than one data set with conflicting preferences for accessing methods. Finally, data set organization constraints tend to propagate *through* computations, because it is most efficient for a computation to write its outputs in the same key order in which it reads its inputs (since that is the order in which the output records will be generated). So, optimization of the type we are considering is necessarily be a problem in global compromise.

The straightforward solution of evaluating the cost of every possible combination of assignments of sort order, device, organization, and access method for data sets and computations in every possible aggregation configuration to determine the least expensive is ruled out by the sheer combinatorics involved. Even with mathematical and special purpose tricks it would be impossibly slow.

To make optimization tractable a heuristic approach must be taken. First different kinds of decisions (e.g. choice of driving data sets, which objects to aggregate) must be decoupled wherever possible. Further decoupling must be judiciously introduced where it is not strictly possible, for the sake of additional simplicity. Such forced decoupling does not mean, though, that decisions that are in fact coupled are treated as if

they were independent. The decoupled decisions are still made with a certain awareness of their effects on other decisions. Finally, as a first order approximation, the optimizer does what is reasonable locally, and then adjusts somewhat for global realities. While we make no claim that this approach will lead to the true optimum, it does produce good and usually near-optimal solutions for real and honest problems.

Stage 4: Code Generation

Stage 4 of Protosystem I consists of the PL/I and JCL Generator modules. The PL/I Generator takes the fully specified output of Stage 3 (the CDSL or Constrained Data Set Language description) as input and produces PL/I code for each job step. This involves the determination and arrangement of PL/I I/O specifics, the construction of the data processing loops, and the programming of the necessary calculations. The JCL Generator then writes IBM OS/360 JCL and ASP instructions for the I/O, administration and scheduling of the compilation and execution of the dps job and job steps.

Conclusion

A model of the data processing system implementation process has been presented and a blue-print, based on that model, for automating the entire process has been developed. Protosystem I is a project to exhibit the feasibility of these ideas. Already, two of the four heretofore manual phases of the software writing process have been automated and are capable of producing acceptable implementations. The automation of the remaining two phases should easily fall within the realm of presently developing technologies within the next decade.

Directions for further investigation include:

expansion of the design repertoire--additional data structures (e.g. hierarchical files, inverted files), the use of Early's iteration inversion ideas, etc.

enlargement of the class of dps's handled (e.g. admitting other types of computations, on-line systems)

development of peripheral automatic technologies--for example, automation of incremental changes to dps's with minimal perturbation/maximal efficiency

automatic development of dps back-up and restart capabilities

Acknowledgements

The author wishes to thank Bill Martin, the originator of many of the ideas in this paper, and Mike Hammer, whose numerous comments, criticisms and suggestions were indispensable.

BIBLIOGRAPHY

1. Balzer, R., "Automatic Programming", Institute Technical Memo, University of Southern California--Information Sciences Institute, 1973.

2. Codd, E. F., "A Relational Model of Data for Large Shared Data Banks", *Communtcattons of the ACM*, 13,6 (June 1970), pp. 377-387.

3. Early, J., "Relational Level Data Structures For Programming Languages", Computer Science Department, University of California, Berkeley, 1973.

4. Hammer, M., Howe, W. and Wladawsky, I., "An Overview of a Business Definition System", *ACM SIGPLAN Nottces*, 9, 4, April 1974.

5. Hawkinson, 1., "The Representation of Concepts In OWL", Fourth International Joint Conference on Artificial Intelligence, Sept. 1975.

6. Nunamaker, J. F. Jr., Nylin, W. C. Jr., and Konsynski, B. Jr. "Processing Systems Optimization through Automatic Design and Reorganization of Program Modules", *Informatton Systems* (Tou ed.), pp. 311-336, Plenum, 1974.

7. Ruth, G. "Automatic Design of Data Processing Systems", Third ACM Symposium on Principles of Programming Languages, Jan. 1976.

8. Ruth, G. "The New Question Answerer", Automatic Programming Group Internal Memo 21, MIT Laboratory for Computer Science, 1975.

9. Sussman, G. "A Computational Model of Skill Acquisition", MIT AI TR-297, MIT Artificial Intelligence Laboratory (August 1973)