Protosystem I—An automatic programming system prototype

by GREGORY R. RUTH

Massachusetts Institute of Technology

Cambridge, Massachusetts

INTRODUCTION

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. Our knowledge and understanding of programming is once again reaching a level where a significant advance in automation is both necessary and possible. In particular, it is now possible (for certain simple application domains) to create a system that will take as input a specification for a user's application and will automatically design and code the desired data processing system. 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—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—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—The system—the actual procedural steps and data representations and organizations—is constructed by intelligent selection from and adaptation of a number of standard implementation possibilities.

Phase 4: Code Generation—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—A form is produced 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.

The products of each stage should 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. 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.

EFFICIENCY ENHANCEMENT IN SYSTEM DEVELOPMENT

To produce a credible and practical result an automatic programming system must perform a reasonable degree of
optimization. Current formal optimization methods pertain mainly to the compilation level; when the entire program development process is automated, new, additional types of optimization will have to be included.

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 (for the sake of I/O efficiency) is something that should be considered during Stage 3 (system implementation) where the data and computational interrelationships among conceptual processing units are most evident. Problems involving efficient loop construction should be handled in a Stage 4.

THE DEVELOPMENT OF PROTOSYSTEM I

The Protosystem I effort began in 1971. Because of the difference in the natures of the technologies involved the work was divided into two parallel efforts: (1) a top-part-of-the-system effort (Phases 1 and 2), essentially of an Artificial Intelligence nature, concerned with natural language comprehension, model formation and problem solving, (2) a bottom-part-of-the-system-effort (Phases 3 and 4) 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). The bottom part of Protosystem I has been completely implemented in the MACLISP language and is operational on the MIT LCS PDP-10. Research and development on the top part, being considerably more ambitious and novel, 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 of the bottom part of Protosystem I is shown in Figure 1. The following sections give an explanation of its workings.

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 data processing system 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. 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 processed in a similar way are termed data sets. In
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 item (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.* If we call these data sets SHIPMENTS-RECEIVED, FINAL-INVENTORY and BEGINNING-INVENTORY, the relationship between BEGINNING-INVENTORY and the others 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

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 input 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

Figure 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 Figure 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

For the dps's which Protosystem I proposes to treat the calculations themselves are easily dealt with; the structuring and manipulation of the masses of data involved forms the major part of the Stage 3 implementation activity. 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 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.

* Thus, a data set is essentially the same as a Codd relation and its keys are what Codd calls primary keys.

** 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 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 configura-
DATA DIVISION

FILE SHIPMENTS-RECEIVED
  KEY IS ITEM
  GENERATED EVERY DAY

FILE BEGINNING-INVENTORY
  KEY IS ITEM
  GENERATED EVERY DAY

FILE TOTAL-ITEM-ORDERS
  KEY IS ITEM
  GENERATED EVERY DAY

FILE QUANTITY-SHIPPED-TO-STORE
  KEYS ARE ITEM, STORE
  GENERATED EVERY DAY

COMPUTATION DIVISION

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

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

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

FINAL-INVENTORY IS BEGINNING-INVENTORY - TOTAL-SHIPPED

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

Figure 3—SSL relational description for the A&T data processing system

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.
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 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 obtained 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 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 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 (whose records have multiple data items) 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.

These access minimizations techniques require that the key order of processing agree in a special way with the organization of the data being processed. Because different computations accessing the same data set may have different preferences for its organization, optimization of the type performed is necessarily 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 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.

ACKNOWLEDGMENTS

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

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