

A PERSPECTIVE ON AUTOMATIC PROGRAMMING

David Barstow

Schlumberger-Doll Research
Old Quarry Road
Ridgefield, Connecticut 06877

ABSTRACT

Most work in automatic programming has focused primarily on the roles of deduction and programming knowledge. However, the role played by knowledge of the task domain seems to be at least as important, both for the usability of an automatic programming system and for the feasibility of building one which works on non-trivial problems. This perspective has evolved during the course of a variety of studies over the last several years, including detailed examination of existing software for a particular domain (quantitative interpretation of oil well logs) and the implementation of an experimental automatic programming system for that domain. The importance of domain knowledge has two important implications: a primary goal of automatic programming research should be to characterize the programming process for specific domains; and a crucial issue to be addressed in these characterizations is the interaction of domain and programming knowledge during program synthesis.

I. INTRODUCTION

Previous work in automatic programming has focused primarily on the roles of deduction and programming knowledge. This paper is concerned with the role played by knowledge of the task domain, a role which seems to be at least as important, both for the usability of an automatic programming system and for the feasibility of building one which works on non-trivial problems. The importance of domain knowledge has two important implications: a primary goal of automatic programming research should be to characterize the programming process for specific domains; and a crucial issue to be addressed in these characterizations is the interaction of domain and programming knowledge during program synthesis.

This perspective has evolved over the last two years during the course of a variety of studies by members of the Software Research group at SDR. These studies will be reviewed briefly, followed by a more detailed discussion of the perspective. Finally, the implications for research directions and methodology will be discussed.

II. THE TASK DOMAIN - QUANTITATIVE LOG INTERPRETATION

The task domain is the interpretation of well logs, an activity central to exploration for hydrocarbons. Oil well logs are made by lowering instruments (called tools) into the borehole and recording the measurements made by the tools as they are raised to the surface. The resulting logs are sequences of values indexed by depth. Logging tools measure a variety of basic petrophysical properties (*e.g.*, the resistivity of the rock surrounding the borehole). Petroleum engineers, geophysicists and geologists are typically interested in other kinds of information which cannot be measured directly (*e.g.*, water saturation — the fraction of the rock's pore space occupied by water rather than hydrocarbons). Log interpretation is the process of deriving the desired information from the measured data.

Log interpretation can be divided into two broad categories: qualitative interpretation is concerned with identifying geological attributes (*e.g.*, lithology — the set of minerals which make up the rock around the borehole), while quantitative interpretation is concerned with numeric properties (*e.g.*, the relative volumes of the minerals). The studies described here have focused primarily on quantitative log interpretation. Quantitative interpretation relies on models — statements of relationships between the measured data and the desired information. These statements may take many forms, such as graphs and equations. For example, the following equation relates water saturation (SW), porosity (P), the resistivity of the water (RW), and the resistivity of the formation (RF):

$$SW^n = \frac{a \cdot RW}{P^m \cdot RF}$$

where a , m and n are parameters that describe certain formation characteristics [Archie 1942]. Many of these relationships have been derived empirically, and the body of relationships generally accepted by expert interpreters is continually changing. The basic domain knowledge for these studies is knowledge of such relationships and their use in computing desired quantities from measured quantities.

III. INITIAL STUDIES

III.A Examination of Existing Software

The first study involved characterizing the nature of existing quantitative log interpretation software. The study was performed by examining, at various levels of detail, several programs in common use by Schlumberger. The programs all shared certain characteristics. They were fairly large, ranging from 50 to 200 pages of FORTRAN, and had been written primarily by experienced log interpreters who had received some training in programming. They had gone through many major revisions as the result of testing and of growth in knowledge about log interpretation. They were intended for relatively wide use on a large number of wells with varying sets of tool readings as inputs. Since each was based on specific models, each program typically embodied significant assumptions about the geological nature of the formation around the well (e.g., that the lithology consists of a sequence of sand and shale layers). The programs were heavily parameterized to enable their application to individual wells with unique characteristics.

In a typical program the code which performs the calculations can be divided into two categories. About one fourth is related to the central calculations of the model. About three fourths deal with the wide variety of special situations that can arise when running the programs (e.g., adjusting inputs and outputs that seem unreasonable according to the assumed model, such as saturation greater than 100%). This division reflects certain characteristics of the domain. Given that one is willing to make assumptions about the formation, the appropriate mathematical models can usually be translated in a straightforward manner into code; thus, the software for the central computation is relatively compact. However, since there is a great deal of uncertainty in selecting from among possible assumptions and models, the software must do extensive testing and adjustment to determine the best possible fit between the data and the assumptions.

The results of this study can be summarized in two observations.

- The complexity of writing quantitative log interpretation software arises from the wealth of knowledge it draws upon, not the algorithms employed.
- Evolution is an inherent feature of quantitative log interpretation and hence a major problem for software developers.

Hence, the most striking conclusion of the study is:

- Algorithm design, the primary focus of most automatic programming research (e.g. [Barstow 1982, Bibel 1980, Smith 1980, Manna and Waldinger 1980]) is far from the most difficult aspect of programming in this domain.

This is probably true of many other domains.

III.B An Experimental System

The second study involved the development of an experimental automatic programming system for a restricted class of quantitative log interpretation software, namely those programs written to test hypothesized models against real data during the trial-and-error process of developing new models. Such models are generally expressed in purely equational terms, and correspond roughly to the central computations of the software studied initially. This class was chosen for several reasons. First, it offered a chance to experiment with some simple kinds of interpretation knowledge (equations) without the need to consider more complex kinds such as heuristics. Second, in-house colleagues might be interested in using it, providing opportunities for testing with "real" users. Third, since the intended users typically spend from hours to days developing test programs, a system which produces programs in minutes would have immediate benefit. The system, called ϕ_0 ("phi-naught"), is described in more detail elsewhere [Barstow et al 1982a, 1982b], so the major features will be discussed only briefly here.

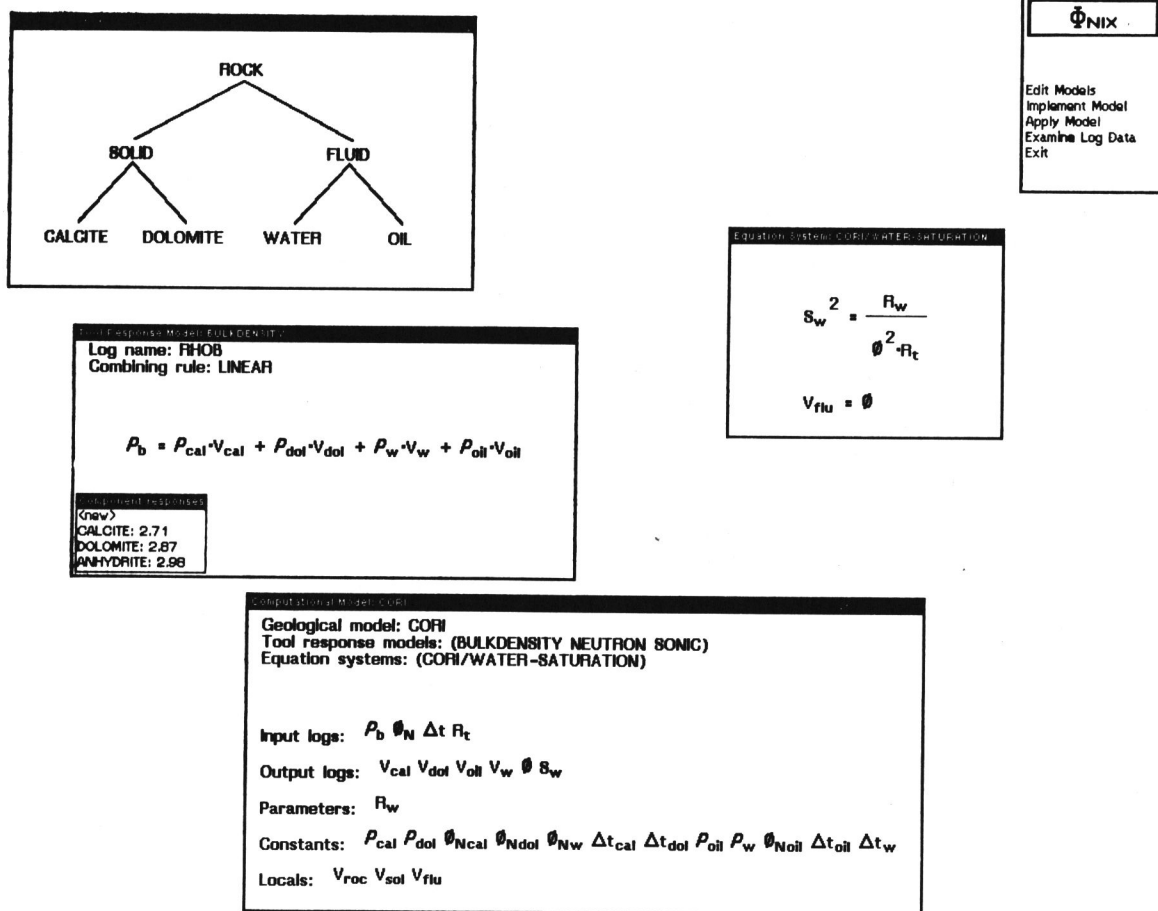
The target user of ϕ_0 is a log analyst concerned with developing a new equational model for use in log interpretation. Such a user may be neither experienced in nor comfortable with traditional computer interfaces. For this reason a great deal of attention has been paid to the user interface. Figure 1 shows the ϕ_0 interface¹ during the process of developing a model, in this case, an unrealistically simple one. The details of the interface are not important for this discussion. Suffice it to say that ϕ_0 exploits the interactive, graphical facilities of the workstation to allow the user to describe assumptions and models using diagrams, notations and non-computational concepts similar to many of the traditional communication mechanisms of log interpretation.

At various times in its development ϕ_0 has written programs in LISP, FORTRAN, and PROSE (a special-purpose language for numeric optimization of equation systems). ϕ_0 's primary mechanisms are algebraic manipulation and syntax-directed translation. Much of ϕ_0 's activity involves traditional programming language techniques.

Although ϕ_0 never came into general use, several observations can be made as a result of building it:

- The impact of an automatic programming system on the work patterns of an experimental scientist can be quite dramatic, since it removes the programming bottleneck between conceptualization and feedback.

1. ϕ_0 currently runs on a Xerox 1100 Scientific Information Processor which includes a 1024 by 808 black and white pixel screen. ϕ_0 is written in Interlisp-D, a LISP dialect with a variety of primitives for dealing with bitmaps, windows, menus, and a "mouse" pointing device.

Figure 1: ϕ_0 Interface

- Good user interfaces are crucial to automatic programming systems. They must provide both an appropriate set of concepts and convenient ways to describe problems using those concepts.
- Special-purpose automatic programming systems often have tight boundaries and are almost useless outside of these boundaries. If a system is going to be useful, it must either have wide boundaries or convenient ways to mix machine- and hand-written code. It may also need to provide access to previously written code.
- If an automatic programming system is expected to produce programs in a variety of languages, an appropriate intermediate language is necessary. However, it is not necessary that it be possible to translate every valid program in that language into every target language. Rather, the intermediate language should provide notations for a wide range of concepts, some of which may be more appropriate for one language than another.
- Algebraic manipulation, such as ϕ_0 's equation solving mechanisms, can be a very powerful tool in developing numeric algorithms. However, such techniques have limits and an ability to use traditional numeric techniques is also necessary. The trade-offs between the two techniques is an area ripe for further investigation.

III.C Case Studies in Program Synthesis

The third study involves developing alternative specifications for existing programs and constructing hypothetical syntheses of programs from them. The programs are generally more complex than those which ϕ_0 could write and involve more domain knowledge. For each of these, the primary goal is to characterize the types of knowledge involved. Preliminary results are available elsewhere [Duffey and Smoliar 1983]; here, the major conclusions to date will be summarized:

- A variety of different types of knowledge play important roles in the development of log interpretation software. These include extensive knowledge of the domain, knowledge of various mathematical formalisms, knowledge of a variety of programming concepts, knowledge of the target language, and even knowledge of the target machine.
- Programming is not a staged process with clear boundaries between the stages and clear limits to the roles played by different kinds of knowledge.
- Knowledge of the domain plays a significant role surprisingly late in the programming process. For example, it is required to select among alternative implementation tech-

niques and to enable the target program to communicate meaningfully with its users.

- Knowledge of the target language plays a significant role surprisingly early in the programming process. For example, if the target language supports numeric solution of implicit equations there may be no need for determining an explicit solution to a system of equations.
- Studying existing software and its documentation is a rich source of insights into the nature of the programming process.

IV. THE ROLE OF DOMAIN KNOWLEDGE

Perhaps the most consistent and important result of these studies has been the realization that knowledge of the domain plays several roles in the activities of an automatic programming system, during both interaction with the user and synthesis of the program.

IV.A User Interaction

Since a user may not be knowledgeable about, or comfortable with, the intricacies of programming systems, he or she must be able to describe problems in a natural way. The system must be able to deal effectively with the terms and notations of the domain. In the case of quantitative log interpretation, the user should be able to use concepts like "porosity," "water," and "shale," and notations like " ϕ_{wb} " (denoting the fraction of a unit of formation that is occupied by water bound to shale). The system should be able to deal with subtle nuances of the domain, for example, that most "shale indicators" are somewhat unreliable. In other words, the user must be able to interact with an automatic programming system in the same way that he or she would interact with another log interpreter.

One crucial part of the interaction between log interpreters is that they share more than just terminology and notations. They also share a great deal of detailed knowledge about log interpretation. They know the responses of the various tools in different materials, they know many mathematical relationships, they are intimately familiar with the interpretation charts which describe relationships diagrammatically, and they have had experience in using common interpretation techniques. When one interpreter describes a technique to another, there is no need to fill in all the details; because of their shared knowledge, the listener will be able to do that. The lesson for automatic programming systems is that they must also be able to fill in the missing pieces. Users will not tolerate interaction in which they must give all of the smallest details. Even if they do tolerate it once, they will certainly not accept the need to repeat the same detail. The system's knowledge base must be capable of evolution, especially for experimental domains such as log interpretation.

In summary, the usability of an automatic programming system will depend largely on the nature of the specification process; the process must be both somewhat informal and highly interactive. These attributes can only be achieved if the system has access to a large and evolving base of knowledge about the domain.

II.B Program Synthesis

One role for domain knowledge during synthesis is to reduce the complexity of the overall task. In the case of log interpretation, a typical FORTRAN program is over fifty pages long; the space of possible programs is enormous. Domain-specific problem-solving heuristics seem to be valuable aids in reducing the space to a more manageable size. As an example, consider the problem of determining the amount of hydrocarbon in a formation. Since hydrocarbons are not uniform (the density of gas varies considerably depending on such factors as temperature and depth), the effect of hydrocarbons on tool measurements is difficult to model precisely, and the models are difficult to deal with mathematically. Log interpreters doing hand calculations have developed a simple heuristic: "Since light hydrocarbons are relatively uncommon, do all of the calculations assuming there are no light hydrocarbons; if the results are implausible, consider the possibility that light hydrocarbons may be present." This heuristic is reflected in code in the form of "porosity analysis" and "hydrocarbon correction" routines. During program synthesis, this domain-specific heuristic enables a complex problem to be reduced to two simpler problems.

Domain knowledge is also needed during synthesis to assist in selecting among alternative implementation techniques. This is perhaps best explained by example. Consider first the problem of representing real numbers: a variety of techniques are possible and selecting an appropriate alternative depends on knowing the range of values and the needed accuracy. As another example, consider the problem of solving a complex system of non-linear equations. From a mathematical viewpoint, this may not be tractable, since an equation may have multiple solutions for the same unknown. However, there may be only one solution with a physically plausible range of values. Finally, consider an equation system which is still too complex to be solved algebraically, even given knowledge of ranges for the terms. In such cases some numeric technique must be employed, usually some form of successive approximation. It is sometimes possible to predict in advance the number of iterations necessary to achieve the accuracy desired. In such cases the loop can be coded with a counter or even open-coded as a sequence rather than a loop. This latter choice was taken in a case in which it was known that the tool readings were only reliable in the first two digits and that two applications of the loop body would achieve four-digit accuracy.

In summary, just as domain knowledge is crucial for the usability of an automatic programming system, it is crucial for the ability of a system to deal with realistic problems. It is needed both for writing large programs and for making appropriate choices among implementation alternatives.

IV.C Implications

In one sense, these observations are simply restatements of generally accepted maxims for building artificial intelligence systems: knowledge is crucial for natural interaction; knowledge can be used to reduce search. The important point is that the observations are concerned with domain knowledge rather than programming knowledge. Automatic programming systems will require large amounts of both kinds of knowledge — it isn't sufficient to build systems which only know about programming.

Of course, if this is true for automatic programming systems, it is probably also true for human programming systems. Among the implications for software engineering, two seem especially important:

- Compilers for general purpose very high level languages may not be able to produce efficient code unless they have some sort of access to knowledge of the domain.
- The traditional separation of software development into specification and implementation may not be feasible unless both specifiers and implementors are knowledgeable about the domain.²

V. THE PERSPECTIVE

These investigations have led to a particular perspective on automatic programming systems, especially those intended for use in real-world situations. This perspective may be summarized by a definition, an assertion, and a conclusion about research goals.

Definition: An automatic programming system is one which allows a computationally naive user to describe problems using the natural terms and concepts of a domain. The description process may involve a degree of informality and imprecision, and the user may omit many details. The system can produce programs which run in a computing environment of the user's choice to effect useful computations.

Assertion: Such an automatic programming system is clearly specific to a particular domain. Knowledge of the domain is crucial to both the usability and the feasibility of such a system.

Research goals: Therefore, a primary goal of automatic programming research should be the development of models of domain-specific programming and techniques for building domain-specific automatic programming systems.

Based on this perspective, several issues seem open for investigation. Some have been addressed, at least in part, in previous research efforts; however, no solid answers have been determined:

- *How can domain knowledge be structured for use by an automatic programming system?* Previous work on structuring domain knowledge has generally been done in the context of natural language understanding and expert systems. It may be that the same structuring could be used by both an automatic programming system and an expert system for the same domain. At this point, there is no direct experimental support for this dual-role hypothesis, although Swartout's work provides some indirect support. [Swartout 1981]
- *How can programming knowledge be structured for use by an automatic programming system?* Previous work in this area has focused on representing relationships between abstract and concrete programming concepts [Barstow 1979, Green 1977, Kant 1981, Kant and Barstow 1981, Rich 1981] and on mechanisms for building complex data structures from simpler ones [Katz and Zimmerman

1981, Low 1978]. It is clear that considerably more remains to be done.

- *What knowledge, either of the domain or of programming, is required to choose an appropriate implementation from a variety of alternatives?* The only previous work on this question [Kant 1981, Katz and Zimmerman 1981] has focused on the required programming knowledge, with domain knowledge used only as a basis for efficiency calculations. The role of domain knowledge in the choice-making process will probably prove to be more significant.
- *How can the interaction between an automatic programming system and the user be modeled?* This is, of course, just an instance of the general issue of human-computer interaction, an issue currently being addressed in a variety of contexts. Among automatic programming systems, ϕ_0 is one of the few for which which it was explicitly assumed that the user may be computationally naive. It is important for many more such case studies to be done.

Unfortunately, the central issue suggested by the perspective, and the key to significant progress in automatic programming, has not been addressed in any substantial way:

- *How can the interaction between domain and programming knowledge during program synthesis be modeled?*

VI. AN EXPERIMENTAL APPROACH

Given the importance of domain knowledge, the best way to address these issues is through experimentation in the context of specific domains. That is, we must develop models of programming for these domains and implement automatic programming systems which test these models. Based on such experiments, we can develop broader models and characterize the utility of different system-building techniques. The validity of this approach as a research methodology clearly depends on characteristics of the individual experiments. It is not sufficient simply to build a variety of application program generators. Unless the domains are suitable and the underlying models of programming are formulated well, it will not be possible to generalize to broader models or to characterize the techniques in a useful way.

While selecting domains and developing models is an art, it is possible to state some guidelines that may be helpful.

- The domain must be non-trivial. There must be considerable room for variability in the class of target problems, the possibility of multiple target languages, or the types of programming techniques which may be employed.
- The model of programming for the domain must clearly characterize the roles played by domain and programming knowledge.
- The model must address the issue of synthesizing not just any program, but a good one.
- The model must be supported by an implemented system intended for real users who need real programs to perform real computations.

Although the development of ϕ_0 has led to some insights, it is too much like an application program generator to provide an example upon which more general systems can be based. Its major weakness is its restricted class of target programs:

2. In a recent paper, Swartout and Balzer discuss this issue at length [Swartout and Balzer 1982].

the class is too simple and there is too little opportunity for variability. ϕ_0 does provide a clear characterization of the roles played by domain and programming knowledge: all of the domain knowledge is embodied in the initial specification phase, with essentially no role for domain knowledge during the synthesis process itself. Unfortunately, it now seems that this strict separation is the wrong characterization.

VII. A PRELIMINARY MODEL OF PROGRAMMING FOR QUANTITATIVE LOG INTERPRETATION

In the remainder of this paper, we will consider an alternative model of programming for quantitative log interpretation, a model which addresses many of the issues suggested above, including the interaction of domain and programming knowledge. This model is speculative, since it is not currently supported by an implementation. At this point, the model's major value is that it is suggestive of the kind of domain-specific model which may form the basis for the development of broader models of programming and more general techniques for building automatic programming systems. For instance, it is plausible that this model may generalize naturally to other types of scientific software.

VII.A Overview

This model of programming for quantitative log interpretation may be summarized as follows:

- Programming involves four qualitatively different types of activities:
 - *Informal problem solving*: This activity is concerned with informal problems involving inputs, outputs, assumptions, and relations stated primarily in domain-specific terms. The problems are informal in that they may be incomplete (not enough information is available to compute the outputs from the inputs) and the input set may not be quite correct (some inputs may be missing, others extraneous). Such problems might be paraphrased in English as "Try to find a way to compute X from Y using relationship Z ." The result of the informal problem solving activity is a set of formal problems stated in terms of any of several different formalisms.
 - *Formal manipulation*: This activity is concerned with problems stated in terms of mathematical formalisms: statistics, analytic geometry, and algebra. The results of these activities are algorithmic statements connected by data flow links.
 - *Implementation selection*: This activity is concerned with selecting from among a variety of implementation techniques for each entity in the algorithmic descriptions produced by formal manipulation. The result of this activity is a complete description of the program in terms of concepts available in the target language.
 - *Target language translation*: The major concern of this activity is the expression of the program in terms of the syntax of the target language. This is a fairly direct translation from the results of the implementation selection activity.
- Each of these activities involves applying transformations based on knowledge of quantitative log interpretation and of programming.

- The overall process of producing a program thus involves applying a sequence of transformations; in such a sequence, the four activities are mixed, but there is a general shift in emphasis from informal problem solving through formal manipulation and implementation selection to target language translation.
- Different sequences of transformations lead to different programs; the programs are all valid implementations of the initial problem description, but may differ with respect to other characteristics, such as target language or efficiency. Knowledge of quantitative log interpretation and of programming is used to determine a sequence leading to a program with desired characteristics.

Let us now consider each of the activities individually followed by the overall synthesis process.

VII.B Informal Problem Solving

An informal problem consists of a set of inputs, a set of outputs, a set of assumptions, and a set of relationships, all expressed in domain terms. For example, an English rendering of one such informal problem is:

Assume the only fluids in the pore spaces in the rock around the borehole are water which flows freely and water which is bound to clay. Try to find a way to compute bound water saturation (SWB) from the resistivity (R) and gamma ray (GR) measurements.

The primary goal of the informal problem solving activity is to determine a set of formal problems which can be put together to solve the overall problem.

This activity has been named "informal" because the problems and subproblems considered during the activity are often incomplete. When inputs and outputs are stated, the intention is that some way of computing the outputs from the inputs is desired. In trying to find an appropriate computation some inputs may be ignored and additional inputs may be considered. Similarly, when relationships or assumptions are given, the intention is that the relationships be used, but there is no requirement that all or only these relationships be used. In other words, the parts of a problem statement provide a kind of focus for the informal problem solving activity.

There are five types of transformations used during informal problem solving:

1. *Reduction to informal subproblems*. These transformations are often suggested by domain-specific heuristics. For example, the following heuristic is relevant to the SWB problem stated earlier: "If you have only unreliable indicators for a quantity, consider each indicator as a separate subproblem, and use some kind of consensus technique to determine a more reliable value." When combined with the fact that R and GR are unreliable indicators for SWB , this would reduce the problem to three simpler ones:

Assume the only fluids in the pore spaces are bound and free water. Try to find a way to compute bound water saturation (SWB) from resistivity (R).

Assume the only fluids in the pore spaces are bound and free water. Try to find a way to compute bound water saturation (SWB) from gamma ray (GR).

Try to find a way to compute a more reliable value for SWB from two unreliable values (from R and GR) using a consensus technique.

2. *Translation into formal problems.* These transformations often involve definitions of domain concepts in terms of domain-independent mathematical formalisms. For example, the following rule could be applied to the GR - SWB problem given above: "If you have a linear indicator for the desired output, the problem may be expressed as a two-point linear relationship between indicator values for the minimum and maximum values of the output." Since GR is a linear indicator for SWB , we have the following formal problem:

Inputs: GR .
Outputs: SWB .
Relationship:

GR and SWB are linearly related with two points being $(GR_{SWB=0}, 0)$ and $(GR_{SWB=1}, 1)$.

3. *Addition of information.* The GR - SWB problem, as stated, is underdetermined: it has one input (GR), one output (SWB), and two other terms ($GR_{SWB=0}$, $GR_{SWB=1}$); since only the value of the input term is known, it is a system with one equation and three unknowns, hence underdetermined. To resolve the difficulty, either additional relationships must be found or more inputs provided. In this case, a reasonable solution is to add two relationships: $GR_{SWB=0}$ is the minimum value of GR in the depth zone of interest, and $GR_{SWB=1}$ is the maximum. Additional information such as this may come from known facts and relationships of the domain or from assumptions the user is willing to make.

4. *Elaboration of general concepts.* Informal problems are often stated in terms of general concepts which do not have precise formal definitions. Before a formal problem can be stated, it is necessary to be more explicit about the desired relationship. For example, the concept of "consensus" covers a broad range of techniques for computing a single value from a set of values, no one of which is especially reliable. Since GR and R are equally unreliable as bound water indicators, and no information about the direction of their unreliability is available, the mean of their values is a good choice. We thus have a formal problem stated in terms of a simple statistical concept:

Inputs: SWB_{GR} SWB_R .
Outputs: SWB .
Relationship:

SWB is the mean of SWB_{GR} and SWB_R .

5. *Introduction of conditionals.* Computed results must often satisfy certain constraints. For example, volumetric results are computed as fractional volumes and must therefore be in the range $[0, 1]$. Such problems are often

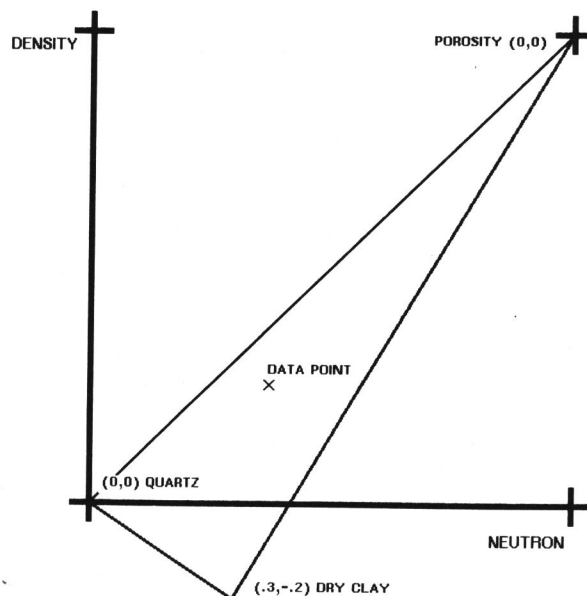


Figure 2: Neutron-Density Coordinate System

described in geometric terms by identifying the points, in a coordinate system whose axes are tool readings, representing 100% concentrations of the materials. Figure 2 shows the points for quartz, dry clay, and porosity in a coordinate system determined by the neutron (N) and density (D) logs. Geometrically, the $[0, 1]$ constraint corresponds to the condition that the data point (i.e., specific values for N and D) fall within the triangle determined by the 100% points. Data points falling outside the triangle represent anomalous situations; identification of such anomalies leads to the introduction of a conditional on the inputs, several new informal problems, and perhaps a request to the user about how the anomalous situation should be handled.

The key ideas behind this formulation of the informal problem solving activity are:

- The heuristics used by expert quantitative log interpreters can provide valuable guidance by suggesting subproblems for consideration.
- There is a large body of quantitative log interpretation facts and relationships which are best expressed in terms of simple statistics, analytic geometry, and algebra.
- Inability to define formal problems suggests incompleteness in the specification which can be resolved by using additional information found in the repository or suggested by the user.
- Analysis in terms of the mathematical formalisms can help identify anomalous conditions as special cases which a target program might encounter when it is run on real data.

VII.C Formal Manipulation

A formal problem consists of a set of inputs, a set of outputs, and a set of relationships stated in terms of mathematical formalisms: statistics, analytic geometry, and algebra. The previous section included examples of formal problems expressed in statistical terms and algebraic terms. The primary goal of the formal manipulation activity is to transform each of the formal problems into an algorithmic form. Three types of transformations are used during this activity:

1. *Reformulation into a different formalism.* The different mathematical formalisms are used by the informal problem solving activity primarily as a matter of convenience. For example, many experimentally determined relationships are represented as charts much more easily than as algebraic relationships. However, unless the target language has mechanisms for dealing directly with such concepts, they must be translated into other formalisms for which the appropriate concepts are available. In the case of quantitative log interpretation, these translations are usually from statistics and analytic geometry into algebra. For example, the volumetric analysis problem described earlier as points in a two-dimensional space may be reformulated as the following system of equations (where DC , Q , P denote the fractional volumes of dry clay, quartz, and porosity):

Inputs: $N D$.
 Outputs: $DC Q P$.
 Relationships:
 $1 = DC + Q + P$
 $N = .3 \cdot DC + P$
 $D = -.2 \cdot DC + P$

2. *Reformulation within a formalism.* Often a problem may be formulated in several equivalent ways within a single formalism. For example, there are many equivalent systems of equations for representing numeric relationships among several terms. Some of these systems may be easier to deal with than others. The problem given above involving three equations in three unknowns may be reformulated as an equivalent system of equations:

Inputs: $N D$.
 Outputs: $DC Q P$.
 Relationships:
 $DC = \frac{N-D}{.5}$
 $P = D + .2 \cdot DC$
 $Q = 1 - P - DC$

This system has the property that, when the equations are considered in order, the terms on the right either are inputs or appeared on the left in an earlier equation. Systems with such an ordering property may be translated directly into algorithms.

3. *Translation into an algorithmic formalism.* The ultimate goal of the formal manipulation activity is to transform the formal problems into an algorithmic representation. Once the appropriate formal manipulations have been done, this is a relatively straightforward process. For example, since the three-equation system given above has the appropriate ordering property, it can be translated directly into a sequence of assignment statements:

Inputs: $N D$.
 Outputs: $DC Q P$.
 Statements:
 $DC \leftarrow \frac{N-D}{.5}$
 $P \leftarrow D + .2 \cdot DC$
 $Q \leftarrow 1 - P - DC$

VII.D Implementation Selection

The result of the formal manipulation activities is a set of algorithmic modules connected by data flow links. The algorithmic modules are described in terms of general computational concepts dealing with control flow, data flow, and operations on three data types: real numbers, sequences, and mappings. For each of these general concepts, several implementation techniques are possible. The goal of this activity is to select techniques for each general concept.³ These selections are guided by two considerations: whether or not the technique is appropriate for the target language, and the relative merit of alternative techniques given some preference criteria (e.g., an efficiency measure).

Three types of transformations are applied during this activity:

1. *Data type refinement.* Most programming languages cannot implement all of the three abstract data types directly; therefore techniques for representing instances of the data types must be selected. For example, two techniques for representing real numbers are floating point representations and ratios of integers. These have different time, space, and accuracy characteristics and the selection of an appropriate technique is clearly guided by the nature of the log measurements and of the computations being done on them.
2. *Data flow refinement.* Most languages have a variety of ways to pass data from one computation to another, and a selection of particular techniques for each of the data flow links must be made. This is an area where different languages may vary significantly. For example, FORTRAN can deal with multiple return values more conveniently than most LISP dialects.
3. *Control flow refinement.* Control flow techniques also differ among target languages. Among the relevant concepts are iteration, recursion, conditionality, and function invocation.

Of course, these types of refinement may be closely linked. For example, deciding to use an array processor to perform similar computations on all elements of a vector has implications for all three types of refinements.

VII.E Target Language Translation

The last activity is a translation from the final algorithmic form into the syntax of the target language. Since the final algorithmic form involves only concepts available in the target language, this is a fairly simple activity. The greatest complexities involve the selection of variable names and producing the program in a readable format.

3. More precisely, a technique must be chosen for each instance of each general concept. For example, there is no reason for all numbers to be represented in the same way.

VII.F The Synthesis Process

Since each activity involves the application of transformations, the overall process can be modeled as a sequence of transformations. In such a sequence the transformations for the four activities are mixed, with a gradual shift in emphasis from informal problem solving through formal manipulation to implementation selection; that is, the process cannot be considered to be simply a rigidly staged sequence of four activities. There are several reasons for mixing the activities. First, work on different parts of the program may proceed at different rates; for example, it may be useful to fully implement one formal problem (e.g., to ensure feasibility) before finishing the informal problem solving activity. Second, work on later activities provides an important kind of feedback on early decisions; this feedback may result in changing such a decision. Third, new informal problems may be introduced during later activities. For example, an optimization strategy to deal with a complex system of equations may require an initial point; the decision to use the strategy is made during formal manipulation, but the initial point determination involves a new informal problem (which may, for example, have less stringent accuracy requirements).

At each stage in a sequence there may be many transformations which could plausibly be applied. Thus, the transformations of the different activities define a space of partially implemented programs; the overall synthesis process is one of exploring this space. At the current time, it is only possible to characterize this space in general terms. Any complete sequence leads to a implementation of the initial problem. Given the informality of the specification, different implementations may have different input/output behavior, although any such differences would be within the accepted bounds of uncertainty in the domain. In addition, there may be considerable variability in terms of efficiency. There is also no guarantee that every sequence of transformations can be completed; there may be dead-end paths.

Since informal problem solving involves many uncertainties, the space probably contains more dead-ends than successful paths. In fact, the major focus of informal problem solving may be simply to find some successful path. There is relatively little concern with finding the best of several possible paths; any one will do. The strategy for exploring this space would include heuristics for selecting transformations and for identifying dead-end paths early. During implementation selection, on the other hand, most of the paths will be successful and more traditional search strategies are probably appropriate. These include simple heuristics based on the target language, quantitative evaluation functions, and explicit construction of much of the space being explored. During formal manipulation, the mechanisms are generally invoked in a goal-directed fashion. The mechanisms are more complex than the transformations of the other activities, and it is often not possible to characterize the situations in which a mechanism will succeed or fail; consequently, the dominant strategy involves trying several mechanisms until one succeeds.

VII.G Characterizing the Knowledge

Given this description of the overall process, it can be seen that knowledge plays two kinds of roles, as the transformations themselves and as selectors of appropriate transformations.⁴ Referring to these two roles as "transformational" and "selective", we may now characterize the roles played by different kinds of knowledge during the four activities.

- Log interpretation.
 - *Facts and relationships derived from theoretical or empirical considerations:* These play a transformational role during informal problem solving by providing information which may be added to informal problems. They also play a selective role during formal manipulation and implementation selection by providing the basic information needed to make certain decisions.
 - *Mechanisms for expressing log interpretation concepts in terms of mathematical formalisms:* These play a transformational role during informal problem solving by producing problems which can be handled by formal manipulation.
 - *Problem-solving heuristics:* These are used during informal problem solving to help reduce problems to simpler problems; in effect, they play both a transformational and a selective role.
- Mathematical formalisms (simple statistics, analytic geometry, algebra).
 - *Taxonomy of concepts:* This plays a transformational role during informal problem solving by indicating specializations for general concepts.
 - *Analytic mechanisms:* These play a selective role during informal problem solving.
 - *Mechanisms for reformulation within a formalism or into a different formalism:* These play a transformational role during formal manipulation by transforming problems into more convenient forms.
 - *Mechanisms for translating a formal problem statement into an algorithmic form.* These play a transformational role during formal manipulation by producing algorithmic modules.
- Programming techniques.
 - *General knowledge of numeric techniques:* This plays a selective role during formal manipulation.
 - *Taxonomy of programming techniques (data flow, control flow, data types):* This plays a transformational role during implementation selection, since it embodies certain refinement relationships.
 - *Mechanisms for analyzing the efficiency of specific techniques:* These play a selective role during implementation selection.
 - *Mechanisms for analyzing the accuracy of numeric representations and techniques:* This is an important special case of the previous two.

4. In terms of traditional search theory, knowledge is embodied both in operators and in heuristics.

- Target language.
 - *Taxonomy of computational concepts available in the language*: This is essentially a sub-taxonomy of the mathematical and programming techniques. It plays a selective role by acting as a filter during formal manipulation and implementation selection.
 - *Mechanism for translating into the syntax of the language*: As the essence of the target language translation activity, this plays a transformational role.

VIII. SUMMARY

Let us now review the basic theme of this paper, overstating slightly for the sake of clarity:

The primary conclusion of our initial studies is that domain knowledge plays a critical role in the programming process. This role is so important that automatic programming systems without considerable domain knowledge will be neither usable by non-computer scientists nor feasible for non-trivial domains. Therefore, a primary goal of automatic programming research should be to develop models of programming which characterize the interaction of domain knowledge and programming knowledge. The best way to achieve this goal is to develop models of programming for specific non-trivial domains, and to test these models by building systems for real users who want real programs that can be run on real data. If these models clearly separate and characterize the roles played by domain and programming knowledge, then we will have the foundation for developing broader models of programming.

ACKNOWLEDGEMENTS

The perspective described here has resulted from the work of many people. Steve Smoliar, Stan Vestal, and especially Roger Duffey have been heavily involved in the design, implementation, and retrospective analysis of ϕ_0 . Steve Smoliar and Roger Duffey have done detailed analyses of existing quantitative log interpretation software as well as several hypothetical syntheses. Paul Barth has been actively involved in the development of the model of programming for quantitative log interpretation described here; Steve Smoliar and Roger Duffey have contributed several key insights during the process. Several of the interpretation developers at SDR have patiently and repeatedly explained the intricacies of log interpretation to us. Bruce Buchanan, Randy Davis, Elaine Kant, Tom Mitchell, and Reid Smith provided valuable feedback on earlier drafts of this paper.

REFERENCES

1. G. Archie, "The electrical resistivity log as an aid in determining some reservoir characteristics", *Petroleum Technology*, 5:1, January 1942.
2. D. Barstow, *Knowledge-based Program Construction*, Elsevier-North Holland, New York, 1979.
3. D. Barstow, "The roles of knowledge and deduction in algorithm design", in J.E. Hayes, D. Michie, Y.-H. Pao (editors), *Machine Intelligence 10*, Ellis Horwood Limited, Chichester, 1982.
4. D. Barstow, R. Duffey, S. Smoliar, S. Vestal, "An automatic programming system to support an experimental science", *Sixth International Conference on Software Engineering*, Tokyo, Japan, September 1982.
5. D. Barstow, R. Duffey, S. Smoliar, S. Vestal, "An overview of ΦNIX ", *National Conference on Artificial Intelligence*, Pittsburgh, Pennsylvania, August 1982.
6. W. Bibel, "Syntax-directed, semantics-supported program synthesis", *Artificial Intelligence*, 12:3, October 1980.
7. R. Duffey, S. Smoliar, "From geological knowledge to computational relationships: a case study of the expertise of programming", (in preparation).
8. C. Green, "A summary of the psi program synthesis system", *Fifth International Conference on Artificial Intelligence*, Cambridge, Massachusetts, August 1977.
9. E. Kant, *Efficiency in Program Synthesis*, UMI Research Press, Ann Arbor, 1981.
10. E. Kant, D. Barstow, "The refinement paradigm: the interaction of coding and efficiency knowledge in program synthesis", *IEEE Transactions on Software Engineering*, 7:5, September 1981.
11. S. Katz, R. Zimmerman, "An advisory system for developing data representations", *Seventh International Conference on Artificial Intelligence*, Vancouver, British Columbia, August 1981.
12. J. Low, "Automatic data structure selection: an example and overview", *Communications of the ACM*, 21:5, May 1978.
13. Z. Manna, R. Waldinger, "A deductive approach to program synthesis", *ACM Transactions on Programming Languages and Systems*, 2:1, January 1980.
14. C. Rich, "Inspection methods in programming", Massachusetts Institute of Technology: AI-TR-604, June 1981.
15. D. Smith, "A survey of the synthesis of LISP programs from examples", *International Workshop on Program Construction*, Bonas, France, September 1980.
16. W. Swartout, "Explaining and justifying expert consulting programs", *Seventh International Conference on Artificial Intelligence*, Vancouver, British Columbia, August 1981.
17. W. Swartout, R. Balzer, "An inevitable intertwining of specification and implementation", *Communications of the ACM*, 25:7, July 1982.