AI Magazine Volume 5 Number 1 (1984) (© AAAI)

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

---

Used by permission of the International Joint Conferences on Artificial Intelligence; copies of the Proceedings are available from William Kaufmann, Inc , 95 First St., Los Altos, CA 94022 USA.

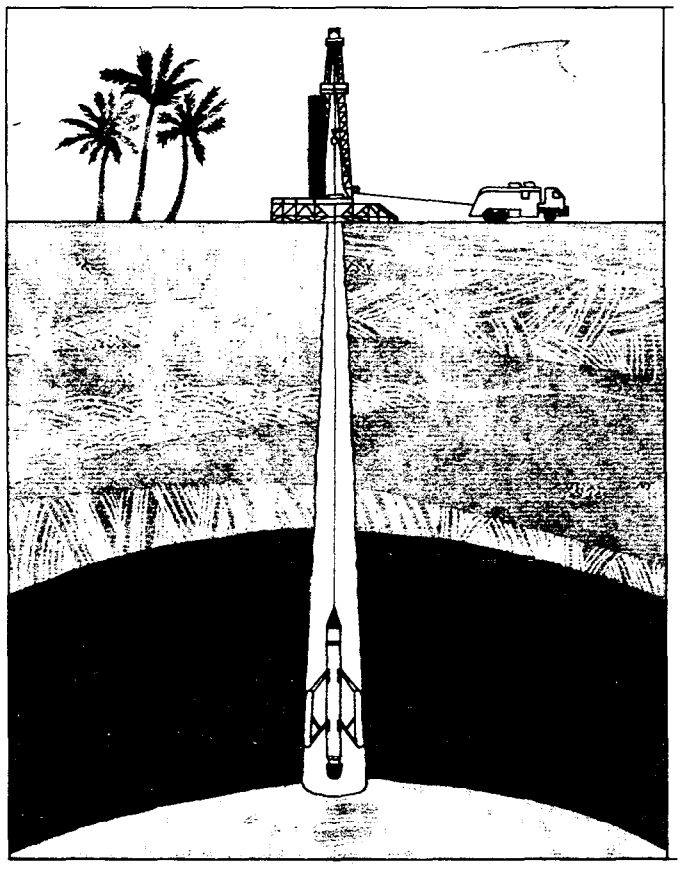
The perspective described here has resulted from the work of many people Steve Smoliar, Stan Vestal, and especially Roger Duffey have been heavily involved in the design, implementation, and retrospective analysis of ϕ_0 Steve Smoliar and Roger Duffey have done detailed analyses of existing quantitative 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 duringrr 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

MOST PREVIOUS WORK in automatic programming has focused on the roles played by deduction and programming knowledge in the programming process. For example, the work of Green (1969) and Waldinger and Lee (1969) in the late 1960s was concerned with the use of a theorem-prover to produce programs. This deductive paradigm continues to be the basis for much research in automatic programming (*e.g.*, Manna & Waldinger 1980, Smith 1983). In the mid 1970's, work on the PSI project (Barstow 1979, Green 1977, Kant 1981) and on the Programmer's Apprentice (Rich 1981) was fundamentally concerned with the codification of knowledge about programming techniques and the use of that knowledge in program synthesis and analysis Work within the knowledge-based paradigm is also continuing (*e.g.*, Barstow 1982, Waters 1981).

This article is concerned with the role played by knowledge of the task domain, a role which seems to be at least as important. One of the reasons for this importance derives from the basic motivating assumption for work on automatic programming: there are many computer users who would prefer not to do their own programming and who would benefit from a facility that could quickly and accurately produce programs for them. The primary concern of these users is not computation — they generally are not interested in the idiosyncrasies of the programming process and certainly don't want to learn the strange notations computer scientists have developed. Rather, they are interested in some application domain — they have problems they wish

solved and questions they wish answered. Computation is merely a tool to help solve the problems and answer the questions. Conventional programming is a hindrance to their use of that tool. It would be much more useful to them if they could communicate in the natural terms, concepts, and styles of their domain. For such interaction to be effective, the automatic programming systems must understand a great deal about the domain. Another reason for the importance of domain knowledge is that the problems to be solved and the questions to be answered are generally so complex that straightforward techniques are inadequate to write programs to solve them. Knowledge of the task domain can play a major role in helping a machine to cope with this complexity.

This perspective on the role of domain knowledge in automatic programming has evolved over the last two years during the course of a variety of studies by members of the Software Research group at Schlumberger-Doll Research [SDR]. These studies will be reviewed briefly, followed by a more detailed discussion of the perspective. An experimental research methodology will be illustrated by a project currently underway at SDR.



Logging an Oil Well.

Figure 1.

The Task Domain: Quantitative Log Interpretation

The task domain is the interpretation of well logs, an activity central to exploration for hydrocarbons. As illustrated in Figure 1, 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. (See Figure 2.) 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). Figure 2b shows a volumetric analysis based on the logs of Figure 2a 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 (S_w), porosity (ϕ), the resistivity of the water (R_w), and the resistivity of the formation (R_t):

$$S_w^n = \frac{a \cdot R_w}{\phi^m \cdot R_t}$$

where a, m and n are parameters that describe certain formation characteristics (Archie 1942). Since the pore spaces must be occupied by either water or hydrocarbons, a low water saturation indicates the presence of oil or gas.

Although the interpretation models themselves are relatively simple, applying them to a particular problem involves a great deal of uncertainty. There are over one hundred qualitatively different lithologies. It's been estimated that it would require over four hundred numeric parameters, such as a, m and n, to fully characterize a formation. Since there are only about a dozen measurements, the situation is hopelessly underdetermined. Consequently, quantitative log interpretaion is a highly expert activity, based not only on a knowledge of a variety of relationships, but also of when and how to use them. This knowledge is the basic task domain knowledge for our automatic programming studies.

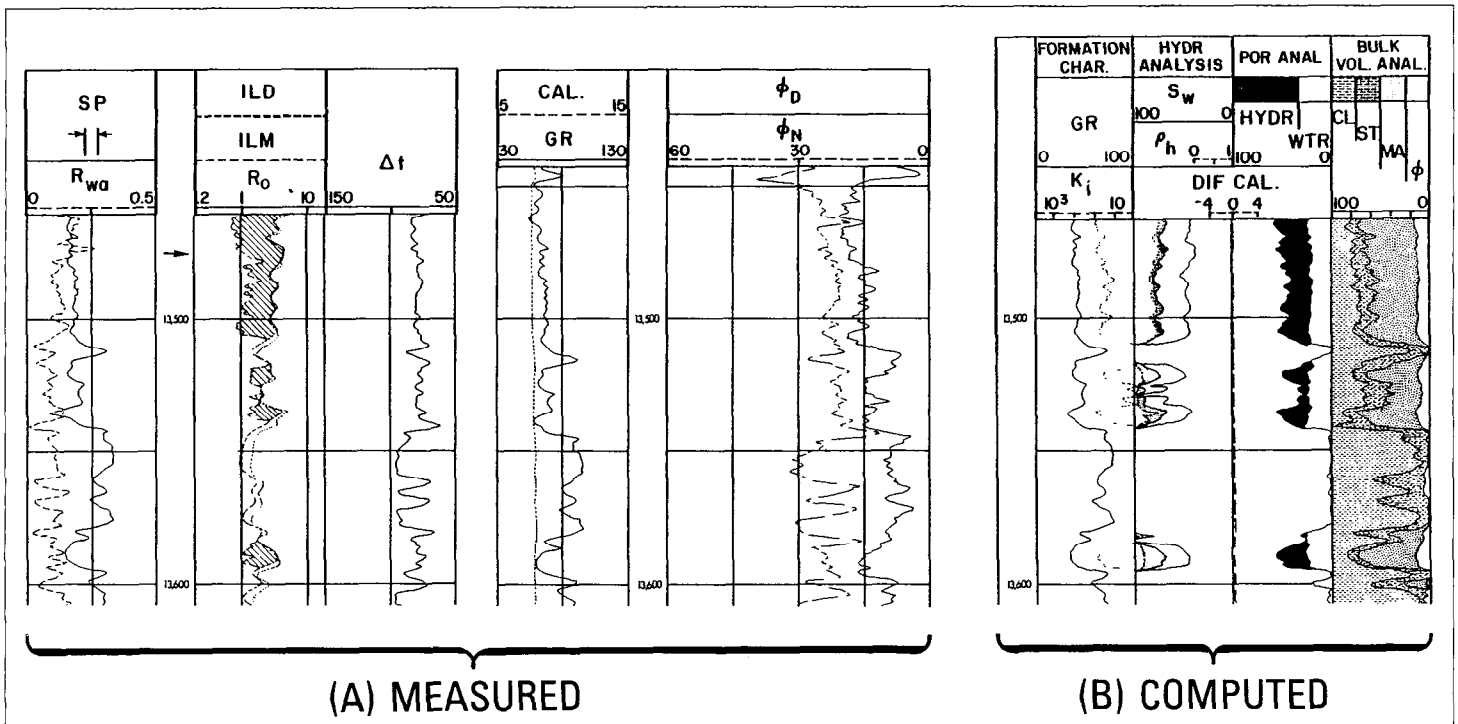


Figure 2 Oil Well Logs.

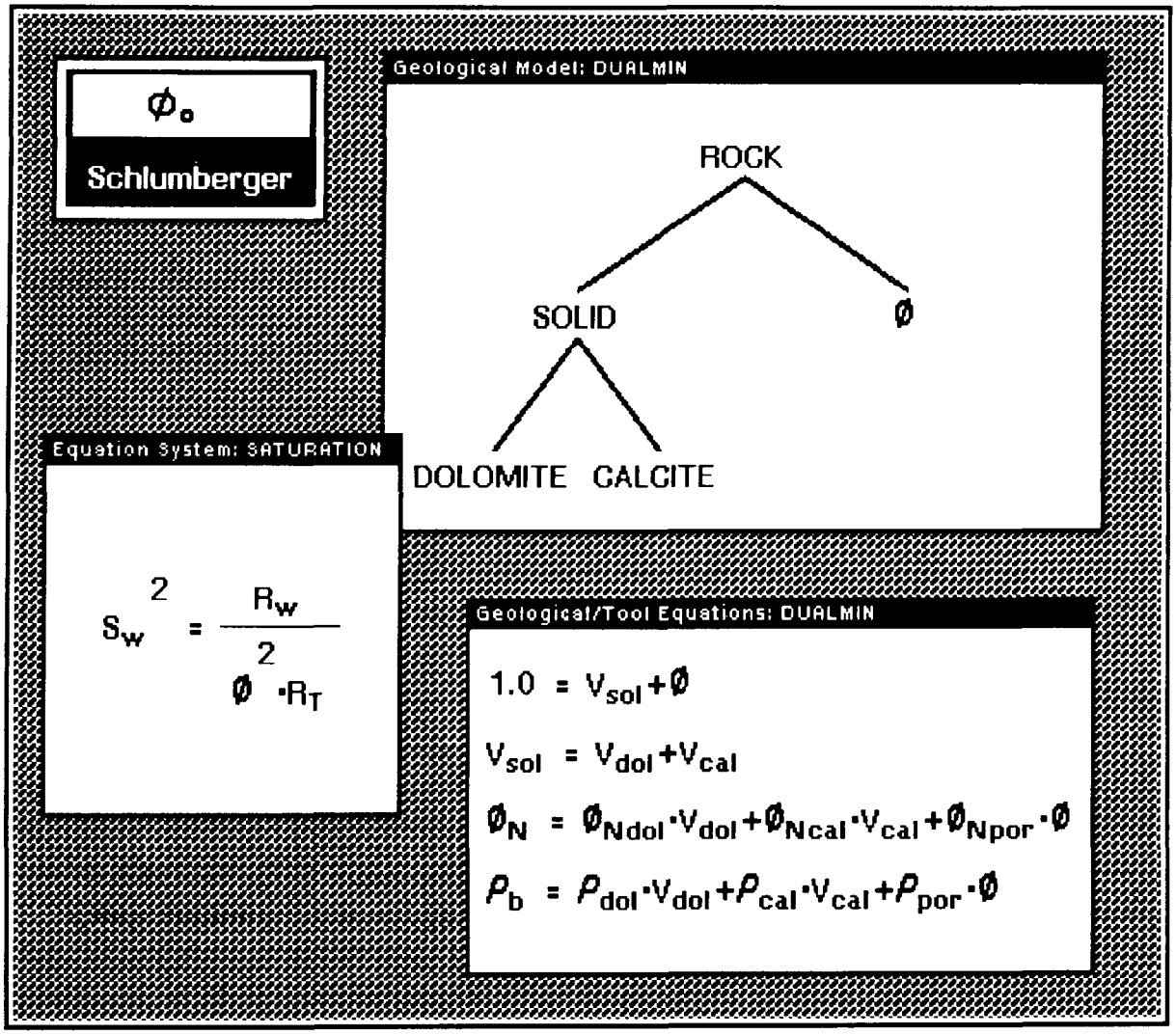
Initial Studies

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 moderate in size, 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. In a way, the special situations are the qualitative side of quantitative log interpretation

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

1. *The complexity of writing log interpretation software arises from the wealth of knowledge it draws upon, not the algorithms employed.* That is, algorithm design, the primary focus of most automatic programming research (*e.g* , Barstow 1982, Bibel 1980, Smith 1980, Manna & Waldinger 1980), is not the hard part of programming in this domain. This is probably true of many other domains
2. *The state of knowledge about log interpretation is constantly changing. New experiments are performed, new data is gathered, new theories are developed, new techniques are tested* That is, evolution is an inherent feature of quantitative log interpretation, and hence a major problem for software developers As with size complexity, this is an area which has not received a great deal of attention in automatic programming research Of course, it could be argued that with fully automatic programming, there is no need to worry about evolution. For each change, the program can simply be rewritten. A counterargument is that, if the automatic programming system knows what it is doing, it should be able to document the program enough that it can make many changes easily.



ϕ_0 Interface.

Figure 3.

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 seconds or minutes would have immediate benefit. The system, called ϕ_0 ("phi-naught"), was implemented in INTERLISP-D which runs the XEROX 1100 series workstations. ϕ_0 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. An explicit goal was that ϕ_0 be easy for such people to use. Toward this end, ϕ_0 includes an icon- and menu-oriented interface incorporating the standard notations and concepts of the domain. Figure 3 shows the ϕ_0 interface during the process of developing a simple model. In this case, the model describes the responses of the neutron (denoted by ϕ_N) and density (ρ_b) tools in a formation consisting of calcite, dolomite, and porosity (The fractional volumes of the three materials are denoted by V_{cal}, V_{dol} and ϕ, respectively. V_{sol} denotes the fractional volume of solid. ϕ_{Ncal}, ϕ_{Ndol} and ϕ_{Npor} denote the characteristic

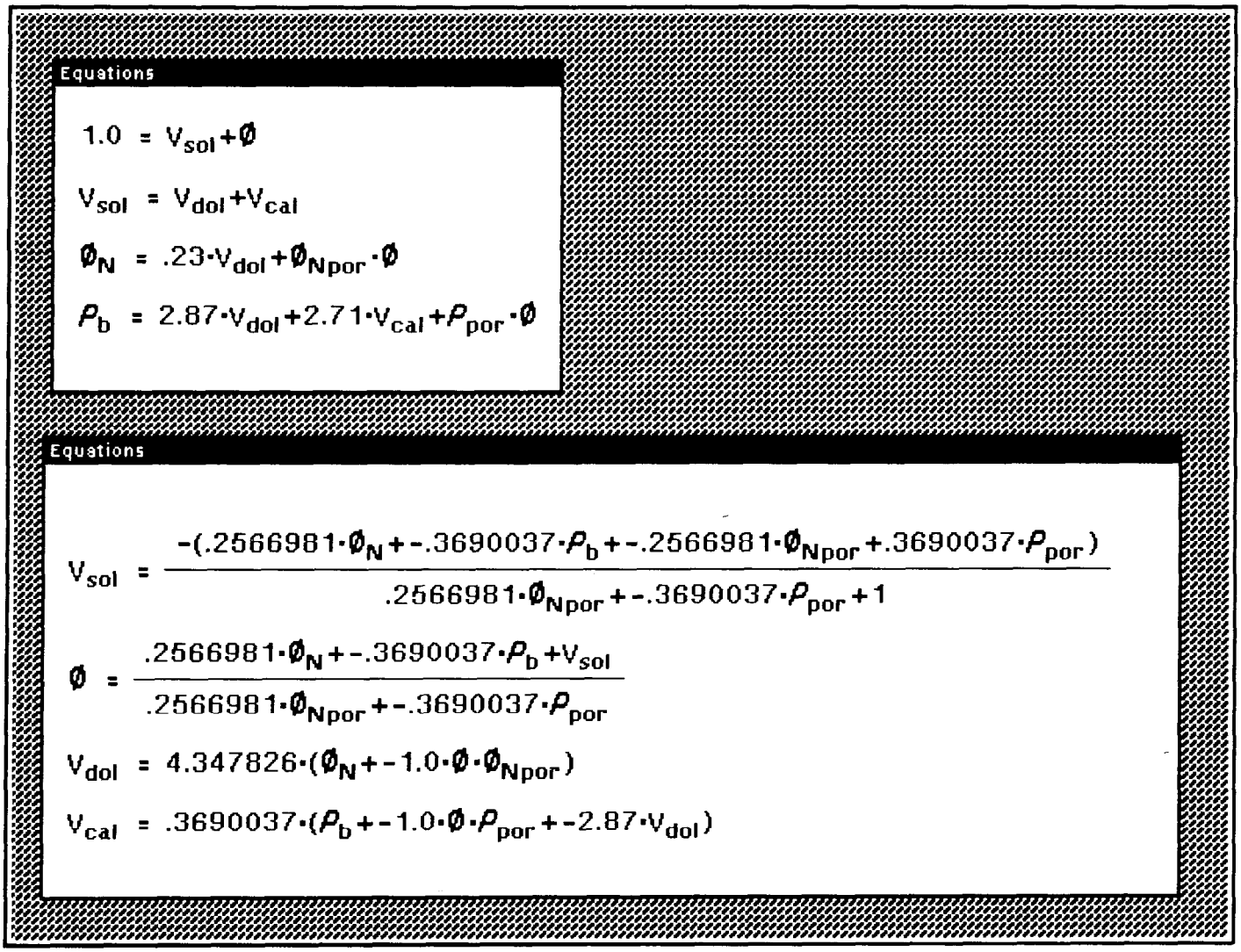
responses of the ϕ_N tool in the three materials; similarly, ρ_{cal}, ρ_{dol}, ρ_{por} denote the ρ_b responses.) Since the characteristic responses of the tools in the materials are known geophysical constants, this model is a system of four equations in six unknowns Hence, it can be used to compute any four of the unknowns given values for the other two. Typically, the tool measurements at some depth in the well are known, and the equation system would be used to compute the fractional volumes of the different materials. Such equation systems consistitute the specifications given to ϕ_0's synthesizer.

At various times in its development ϕ_0 has written programs in several different target languages. For LISP and FORTRAN, the basic technique was to use an algebraic manipulation system to turn the implicit system into an explicit one. At each step, the size of the system is reduced by solving one equation for one term and substituting the result into the remaining equations. Figure 4 shows both the initial system and the solved system.[1] If the equations are considered in order, the terms on the right of each equation are either inputs or appear on the left of an earlier equation. In other words, they may be considered to be a sequence of assignment statements.[2]

Not all systems of equations can be solved explicitly. ϕ_0 used PROSE (1979), a mathematical package which uses numeric techniques to solve implicit systems of equations, as

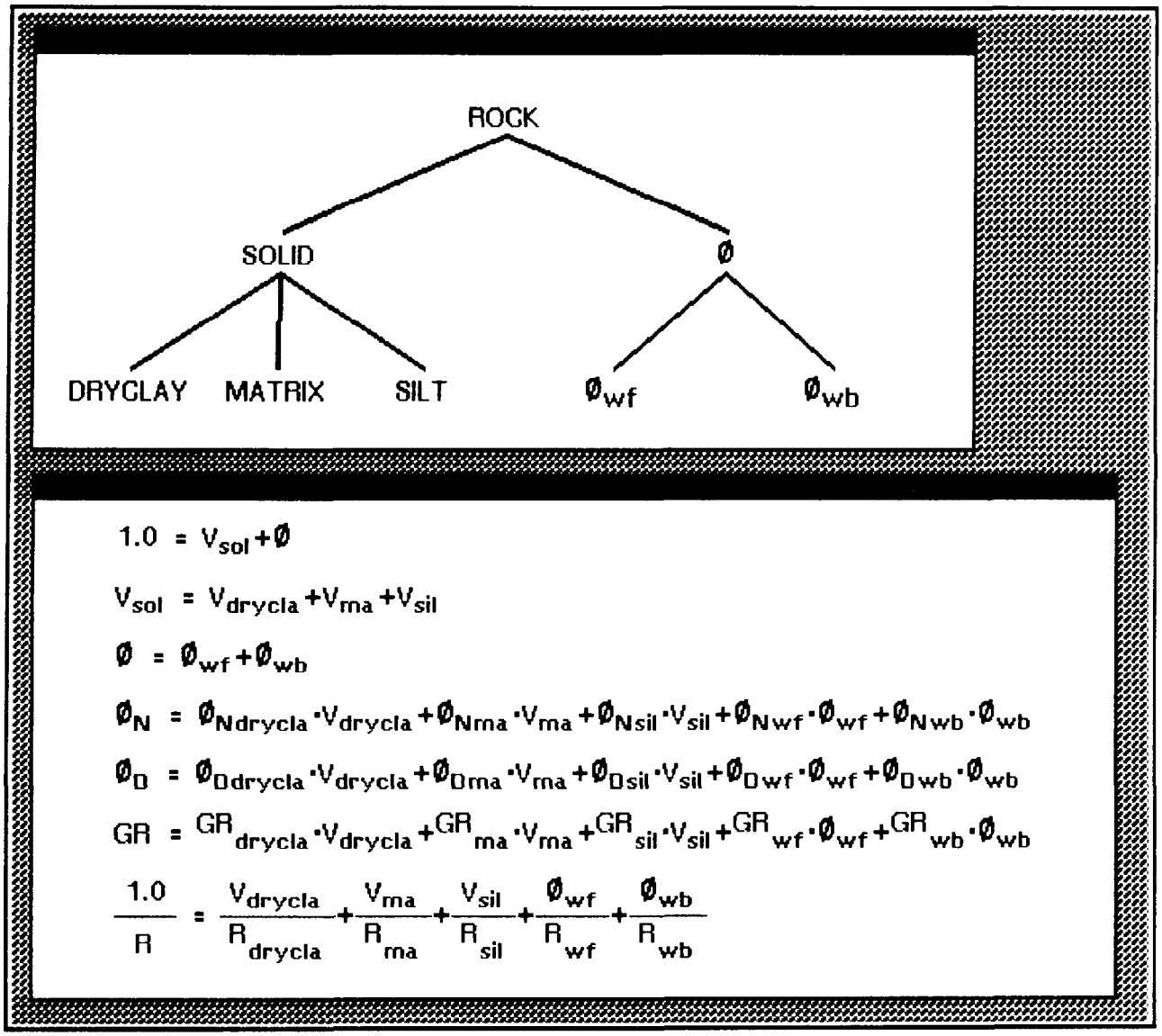
[1]Appropriate values have been substituted for the geophysical constants, except that the responses of the tools in porosity (ϕ_{Npor} and ρ_{por}) have been left as run-time parameters which the user may adjust to reflect the nature of the fluid in the pore spaces

[2]MACSYMA [Bogen et al 1975] has a similar facility for producing FORTRAN statements



Algebraic Solution of an Equation System.

Figure 4.



Hypothetical Synthesis: Initial Statement.

Figure 5.

a target language for those cases which could not be handled algebraically.

Several observations can be made as a result of the ϕ_0 experience:

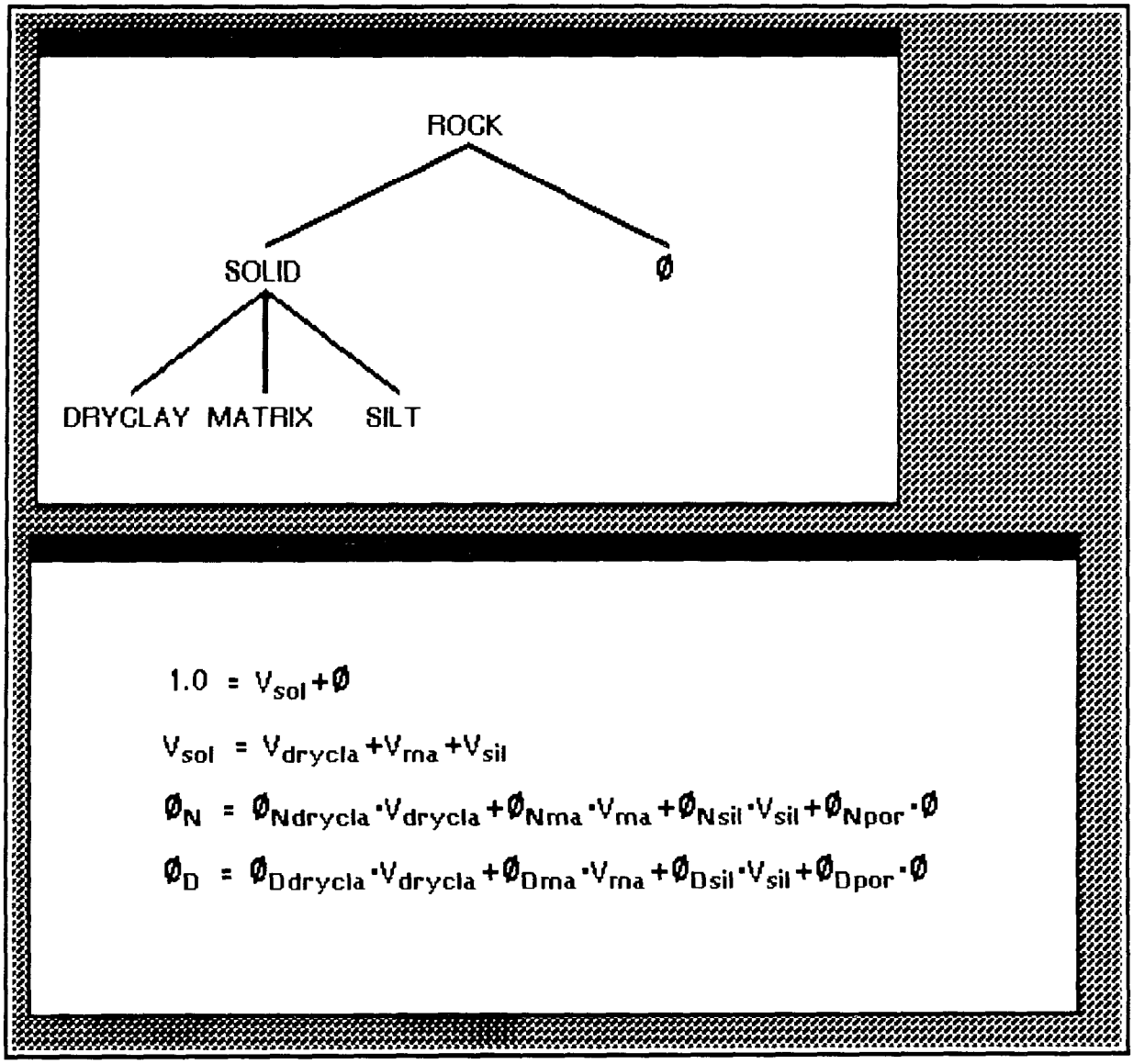
- *Good user interfaces are crucial to automatic programming systems They must provide both an appropriate set of concepts and convenient ways to describe problems using those concepts.* This statement may be obvious, but is worth making quite explicit
- *Special-purpose automatic programming systems 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.* ϕ_0 never came into widespread use, primarily because its boundaries were too narrow The target class, equational interpretation models, was too restricted In particular, certain procedural concepts (such as sequencing and conditionality) seem to be natural for some aspects of quantitative log interpretation; these concepts were outside the bounds of ϕ_0's capabilities
- *Algebraic manipulation is quite powerful, but has obvious limitation* Not all systems can be solved explicitly, and there are some questions about the numeric precision and stability of the computations resulting from the explicit solution of large systems Domain knowledge may enable some of these limitations to be overcome. As a simple example, knowing the plausible range of values for terms in a quadratic equation may enable the selection of one of the two possible roots. This is an area ripe for further research

Hypothetical Syntheses. The third (and on-going) study involves the construction of detailed, step-by-step descriptions of the process of writing particular programs This methodology has provided useful insights in the past (*e.g.*, the classical work by Floyd (1971), Manna and Waldinger (1975), and the preliminary work on PSI (Green & Barstow 1977) and the Programmer's Apprentice (Rich & Shrobe 1978).) The key to the methodology is to pick good examples to work on. In our case, we are working on complete log interpretation programs, not just the central computations which ϕ_0 could deal with. For each such synthesis, the primary goal is to characterize the types of knowledge involved. In this section, a few steps from one hypothetical synthesis will be discussed.[3]

The problem under consideration is to perform a volumetric analysis assuming that the rock is composed of three materials (dry clay, silt, and matrix consisting of quartz) and that the fluid consists of two types of water (free water (ϕ_{wf}) which flows freely in the pore spaces; and bound water (ϕ_{wb}) which is chemically attracted to clay). Measurements from four tools are available: neutron (ϕ_N), density porosity (ϕ_D), gamma ray (GR), The problem and the associated set of seven equations in seven unknowns are shown in Figure 5.

From a mathematical point of view, the system could be solved algebraically and translated directly into code. However, log analysts know that the GR and R_t equations are only rough approximations — they're not very reliable. A better way to write this program is to use a traditional interpretation heuristic — separate out the solid analysis from

[3]Results for other syntheses are available elsewhere (Duffey & Smoliar 1983)



Hypothetical Synthesis: Solid Analysis.

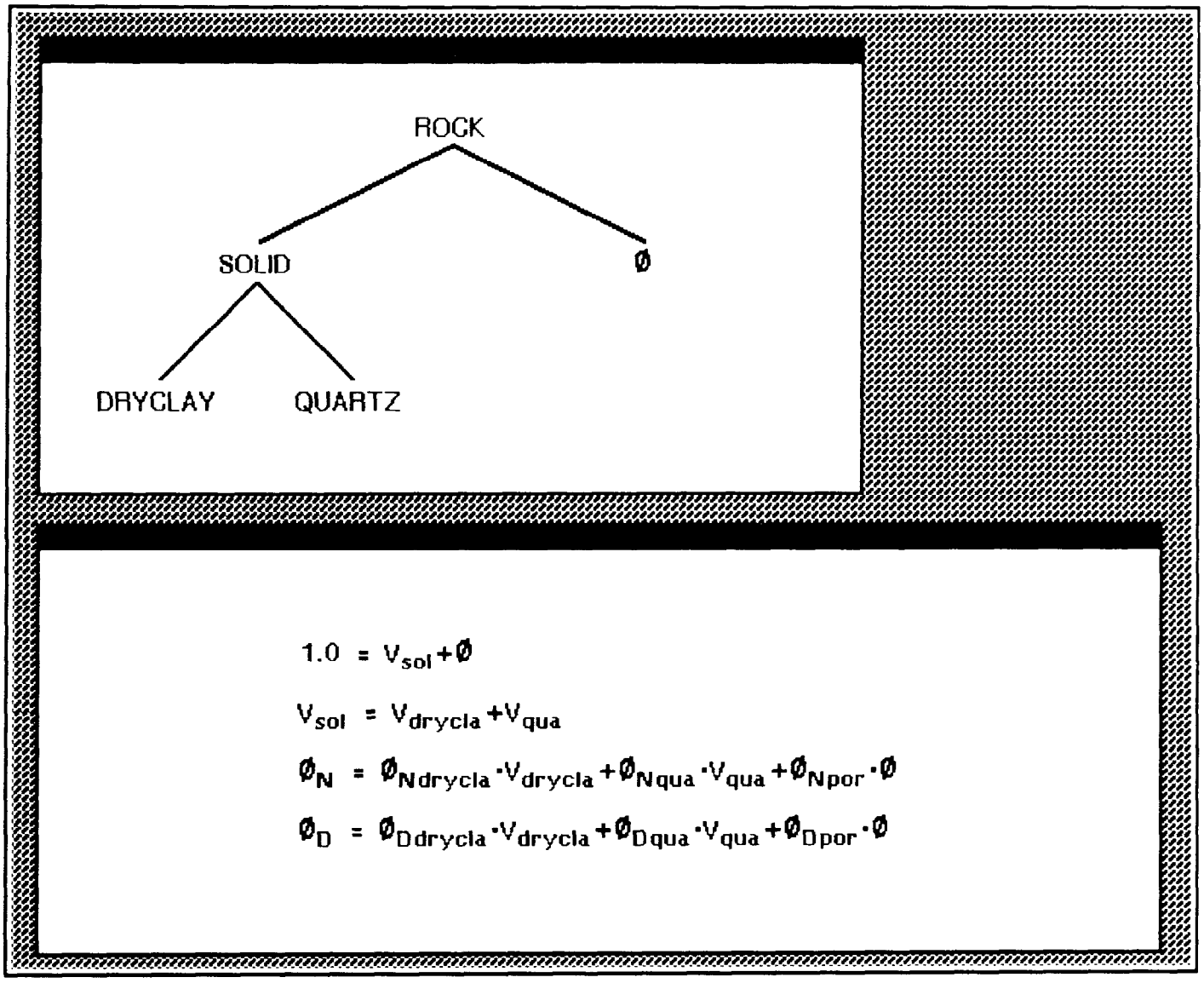
Figure 6

the fluid analysis, using whichever measurements are most appropriate for each. The resulting solid analysis part involves three materials but only two measurements, as shown in Figure 6.

We now have a system of reliable equations but it is underdetermined (four equations in five unknowns). Once again, knowledge of log interpretation resolves the problem. Since the matrix is made up of quartz, and silt is essentially ground up quartz, the responses of the two tools in matrix and silt are identical. Given only these two measurements, it would be impossible to distinguish between matrix and silt. Thus, the solution is to break the problem into two parts: one is concerned with porosity, quartz, and dry clay; the other is concerned with somehow distinguishing between matrix and silt. The first of these is a system with four equations and four unknowns, as shown in Figure 7

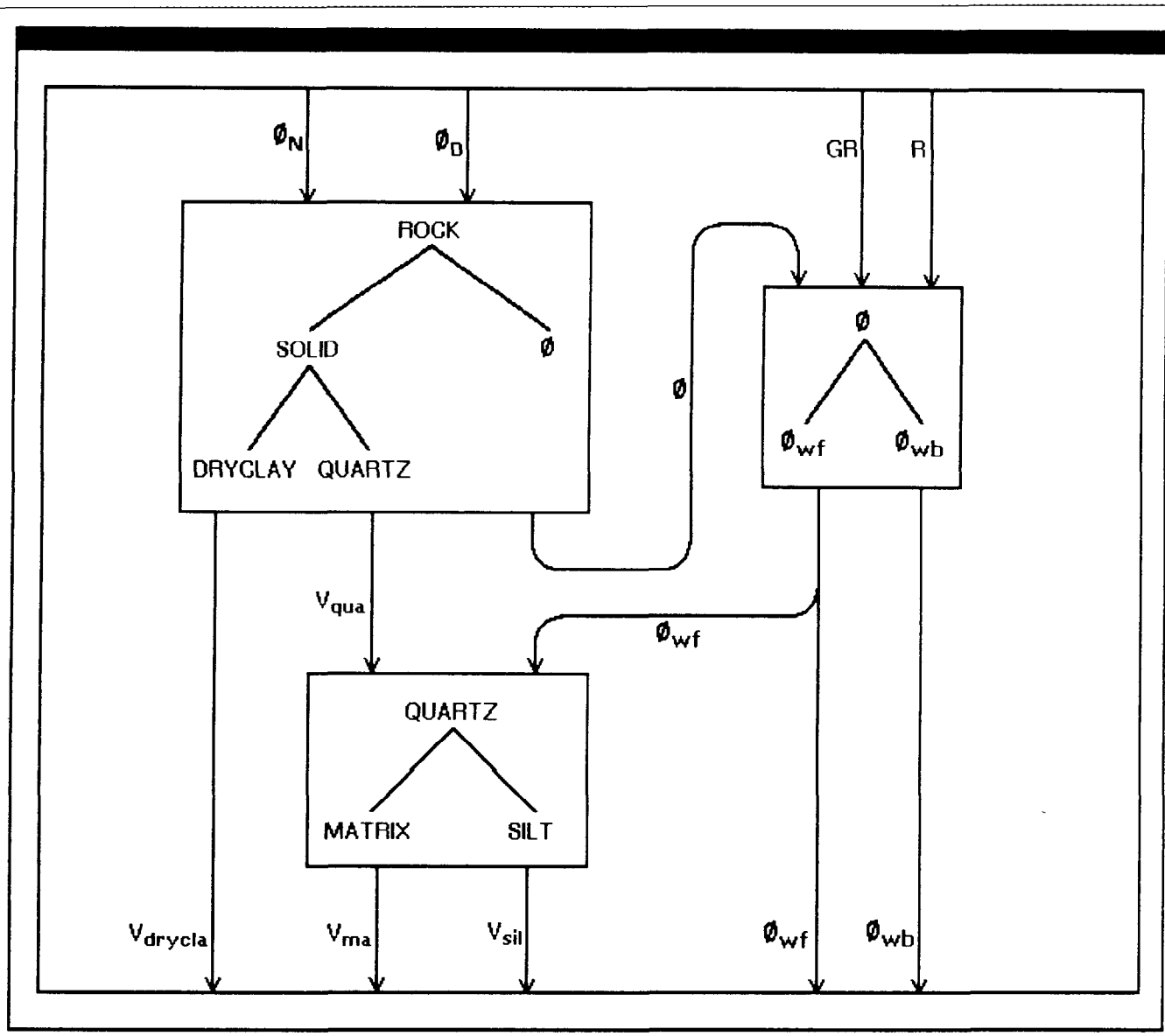
At this point, the problem might seem to be one of straightforward algebraic manipulation. Recall, however, that explicitly solving a system of equations involves several choices: each step involves selecting a term and equation to solve. Such choices can be influenced by a variety of factors. For example, if the target architecture has several processors, it may be possible to take advantage of natural parallelism in the computation. Figure 8 shows a flow chart of the major data paths in the problem. Note that porosity (ϕ) is computed in one block and used in another. The second block could be computed on a second processor as soon as porosity is available. Thus, the equation system should be solved in such a way that porosity is computed first

As these few steps suggest, many different types of knowledge play important roles in the development of log interpretation software. These include extensive knowledge of



Hypothetical Synthesis: Simplified Solid Analysis.

Figure 7.



Hypothetical Synthesis: Data Flow Diagram.

Figure 8.

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. In fact, this is what makes programming such a fascinating process to study — a wide variety of knowledge is used in a wide variety of ways.

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.

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}". 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.[4]

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. This role for domain knowledge might be considered a nice extra that could be ignored — simply provide a sufficiently high-level language for the user. The problem is that "No matter how high the level, it's still programming" (Smoliar & Barstow 1983). Unless that fundamental fact is changed, a wide class of users would still have to learn programming in order to benefit from such systems. Access to a body of domain knowledge is one way to bring about such a change.

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 polynomial 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.

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:

1 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.

2 The traditional separation of software development into specification and implementation may not be feasible unless both specifiers and implementors are knowledgeable about the domain [5]

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 allows a computationally naive user to describe problems using the natural terms and concepts of a

[4] This argument is not new; Balzer *et al* (1977) made a similar argument several years ago

[5] In a recent paper, Swartout and Balzer discuss this issue at length (Swartout and Balzer 1982)

domain, with informality, and imprecision and omission of details. An automatic programming system produces programs which run on real data 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?* A great deal of work on representing domain knowledge has been done in the context of expert systems. It would be a satisfying result if the same knowledge structured in the same way could be used by an automatic programming system The idea is reminiscent of Green's early work (1969), in which the same axioms were used both to solve problems and to write programs, but it has yet to be tested in a complex domain.
- *How can programming knowledge be structured for use by an automatic programming system?* Work on this issue has been proceeding on two fronts One is the relationship between abstract and concrete programming concepts. Notable work here includes the PSI project (Barstow 1979, Green 1977) and the Programmer's Apprentice project (Rich 1981). The other front involves techniques for constructing complex data structures. The work of Low (1978) and Katz and Zimmerman (1981) fit into this category Overall, the groundwork for this issue has been laid What is now required is rather methodical codification of many specific programming techniques.
- *What knowledge is required to choose an appropriate implementation from a variety of alternatives?* Here the question is essentially one of efficiency in the target program It is interesting to note that Simon, in his description of the Heuristic Compiler in the early 1960's, explicitly recognized that the efficiency of the target code was an issue and also explicitly chose to ignore it (1963). There was her reference to the issue in the automatic programming literature for over a decade Finally, Low (1978) considered it in his work on data structure selection More recently, Katz and Zimmerman (1981) looked at it in the context of a data structure advisor and Kant (1981) considered the question in the context of the PSI project. This work is a good start, but the issue deserves much more attention than it has gotten. Our studies suggest that the use of domain knowledge will prove crucial to addressing it well In fact, Kant's work actually characterized one kind of domain knowledge that's needed for selecting the right alternative — knowledge of the plausible values and typical set sizes.
- *How can the interaction between an automatic programming system and the user be modeled?* In one sense, this is just a special case of human-computer interfaces, a topic which has been receiving a considerable amount of attention lately. With respect to the interface, there's probably nothing special about automatic programming systems. In another sense, however, this gets right to one of the core questions of artificial intelligence — how can two knowledgable agents communicate? Automatic programming systems provide a fruitful context in which to explore this question.

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?*

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. By considering ϕ_0 in light of these guidelines, we may see why it is not particularly useful as a basis for more general models:

- *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.* ϕ_0's domain, purely equational interpretation models, was too simple Some extensions which would have increased the complexity are: non-equational concepts; computations over zones in a well, rather than single levels; and some notion of the reliability of the stated relationships. ϕ_0 could write programs in several target languages, although the technique was essentially a "big switch." ϕ_0's repertoire of programming techniques was rather limited

- *The model of programming for the domain must clearly characterize the roles played by domain and programming knowledge.* ϕ_0 provides 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.
- *The model must address the issue of choosing from among several alternative implementations.* For a given target language, there was essentially only one style of implementation.
- *The model must be supported by an implemented system intended for real users who need real programs to perform real computations* As noted earlier, although ϕ_0 was intended for real users and real programs, it never came into widespread use, primarily because of limitations on its target class.

ϕ_{NIX}

As an illustration of an experiment which seems to fit the guidelines, we will consider some of the details of ϕ_{NIX}, another project currently underway at Schlumberger-Doll Research, the scope of which includes full-fledged quantitative log interpretation programs.[6] This class has considerably more variability than ϕ_0's for three reasons. First, it includes non-equational relationships such as charts, tables, and simple procedural concepts. Second, it includes computations over entire wells, not just the single level computations which ϕ_0 dealt with. Third, the target language and machine include simple kinds of parallelism.

To date, we have outlined a model of programming for this target class. The model addresses the two issues noted earlier: the interaction of domain knowledge and programming knowledge; and the selection of an appropriate program from a set of valid alternatives. However, the model is only speculative — it is neither tested nor supported by an implementation. Rather, it must be considered to be a proposal. Nonetheless, the model provides a good example — it suggests the kind of domain-specific model of programming which may generalize to other domains. For example, we hope that this model will generalize to other types of scientific software.

Overview of the Model of Programming. According to this model, programming for quantitative log interpretation involves four 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. We will now consider each of the activities and the kinds of transformations involved in them.

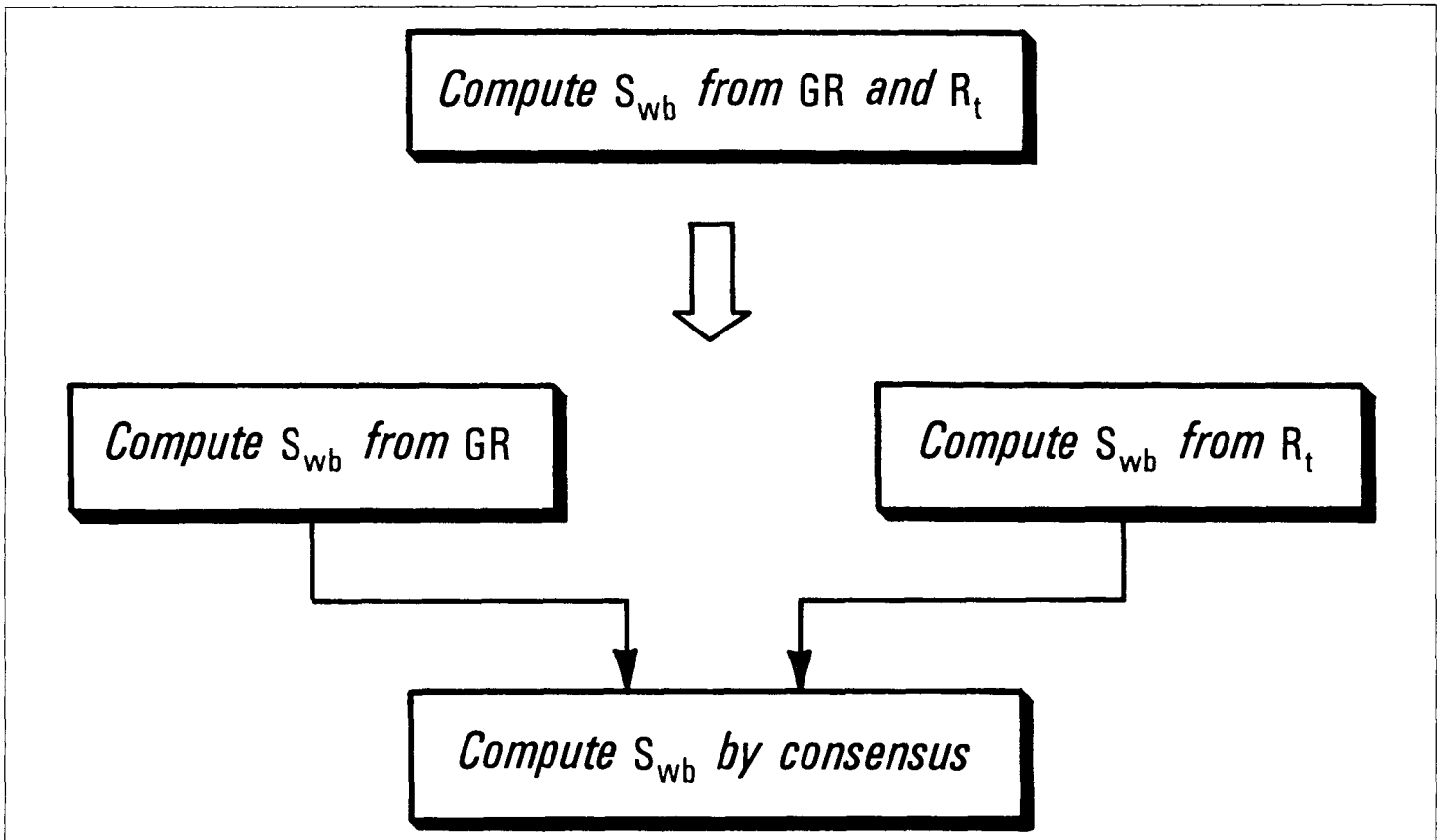
Informal Problem Solving. An informal problem consists of a set of inputs, a set of outputs, a set of assumptions, and a set of relationships, all expressed in domain terms. For example, an English rendering of one such informal problem is:

> 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 (S_{wb}) 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 focus for, but not a restriction on, the informal problem solving activity.

[6] An early report on ϕ_{NIX} provides additional background (Barstow et al 1982b)



Informal Problem Solving: Reduction to Informal Subproblems.

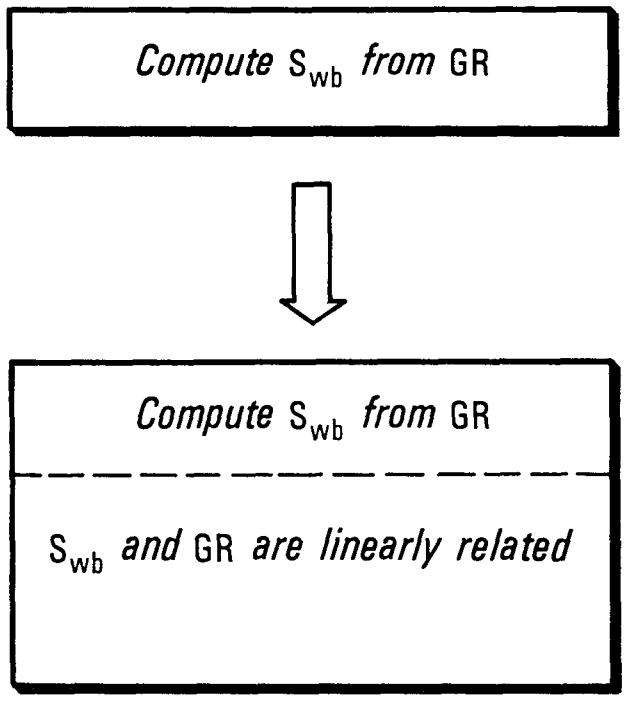
Figure 9.

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 transformation shown in Figure 9 is based on the following heuristic: "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." The result is three simpler informal problems

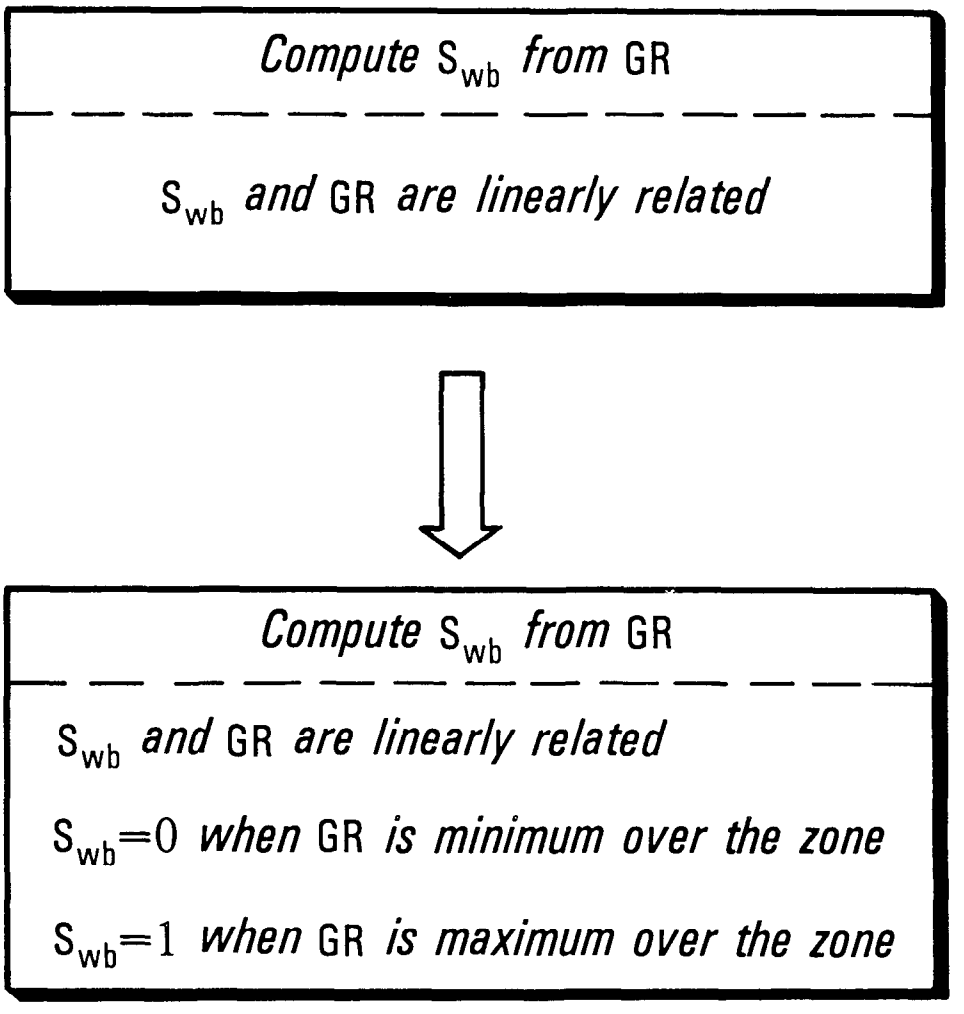
2 *Translation into formal problems.* These transformations often involve definitions of domain concepts in terms of domain-independent mathematical formalisms. For example, the transformation shown in Figure 10 is based on the following rule (and the fact that GR is a linear indicator for S_{wb}): "If 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." The result is a problem stated in terms of formal concepts.

3. *Addition of information* The formal problem, as stated, is underdetermined: it has one input (GR), one output (S_{wb}), and two other terms ($GRS_{wb=0}$, $GRS_{wb=1}$); since only the value of the input term is known, it is a system with one equation and



Informal Problem Solving:
Translation into Formal Subproblems.

Figure 10.



Informal Problem Solving: Addition of Information.

Figure 11.

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, as shown in Figure 11 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_t 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 This transformation is illustrated in Figure 12

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 volumetric problems are often described in geometric terms by identifying the points, in a coordinate system whose axes are tool readings, representing 100% concentrations of the materials. Figure 13 shows the points for quartz, dry clay, and porosity in a coordinate system determined by the neutron (ϕ_N) and density-porosity (ϕ_D) logs Geometrically, the $[0, 1]$ constraint corresponds to the condition that the data point (*i.e* , specific values for ϕ_N and ϕ_D) fall within the triangle determined by the 100% points Data points falling outside the triangle represent anomolous 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 anomolous situtation should be handled.

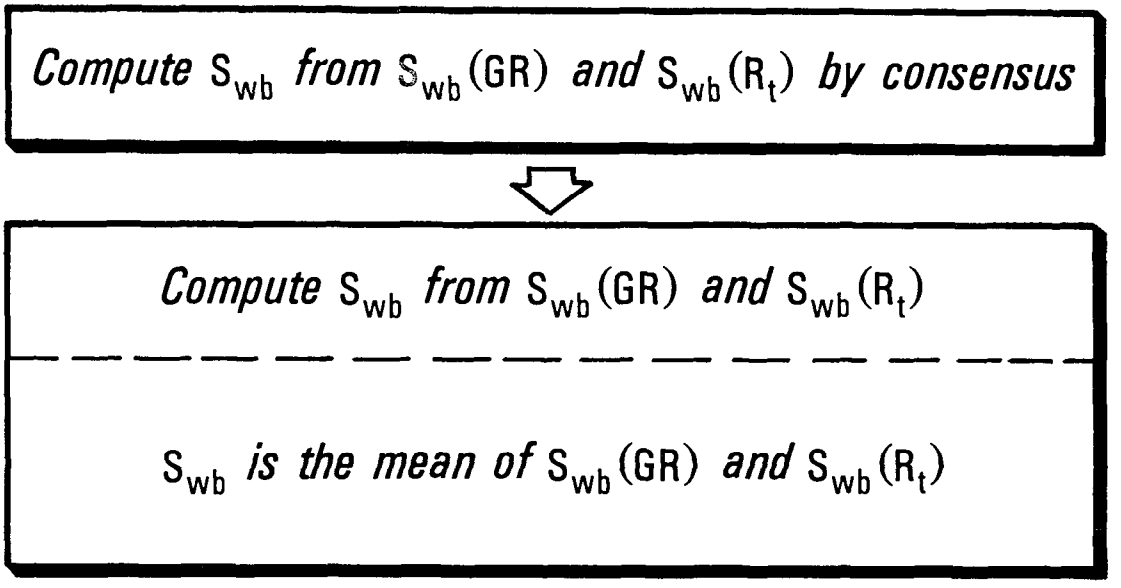


Figure 12. Informal Problem Solving: Elaboration of General Concepts.

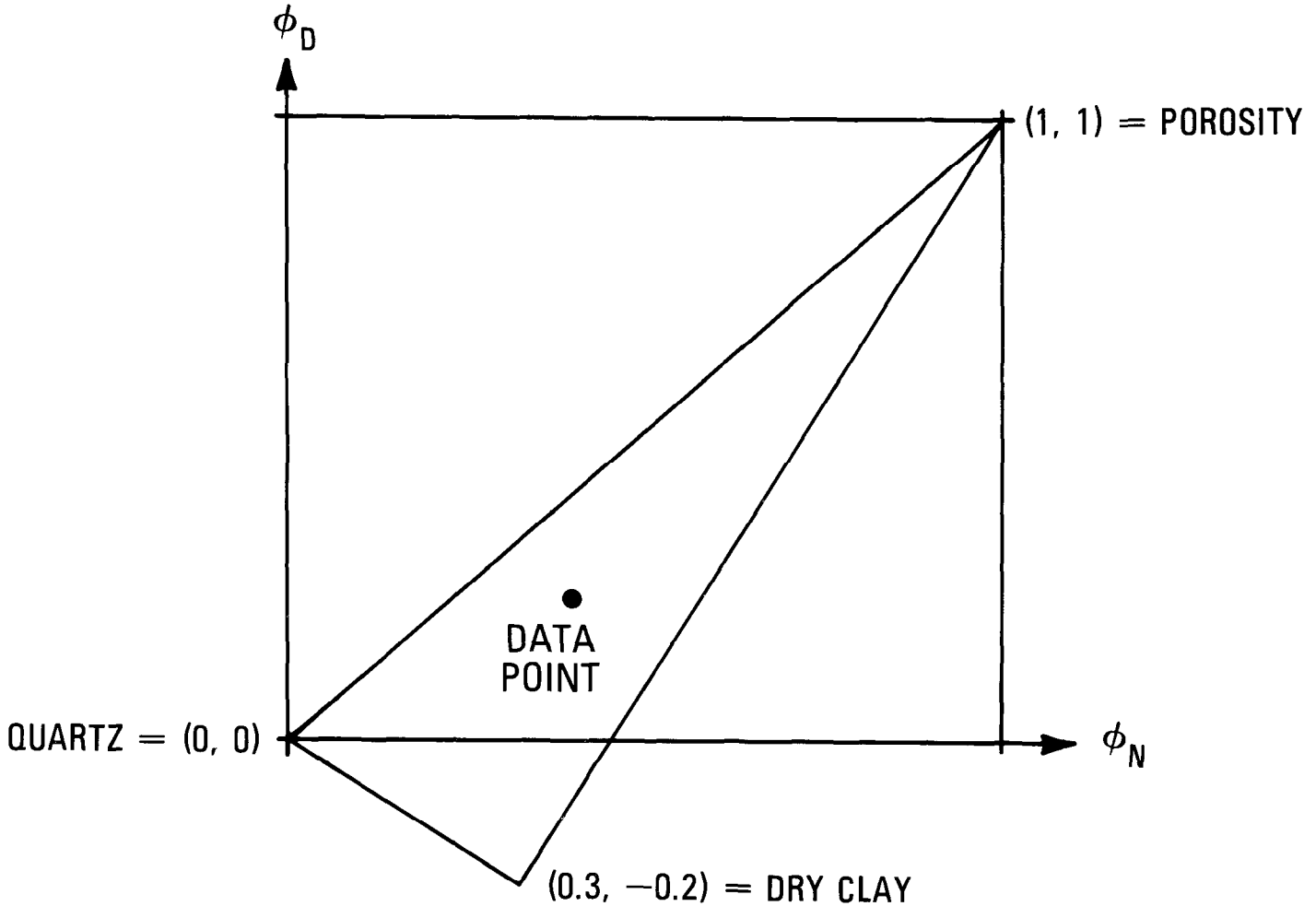


Figure 13. Geometric Representations of a Volumetric Analysis Problem.

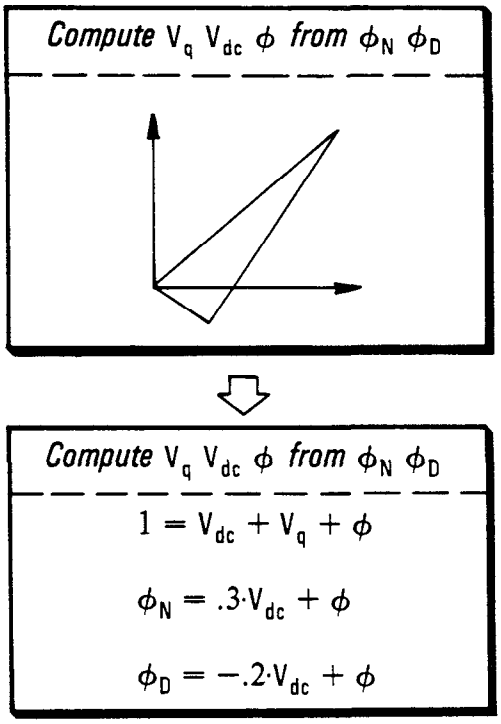


Figure 14. Formal Manipulation: Reformulation into a Different Formalism.

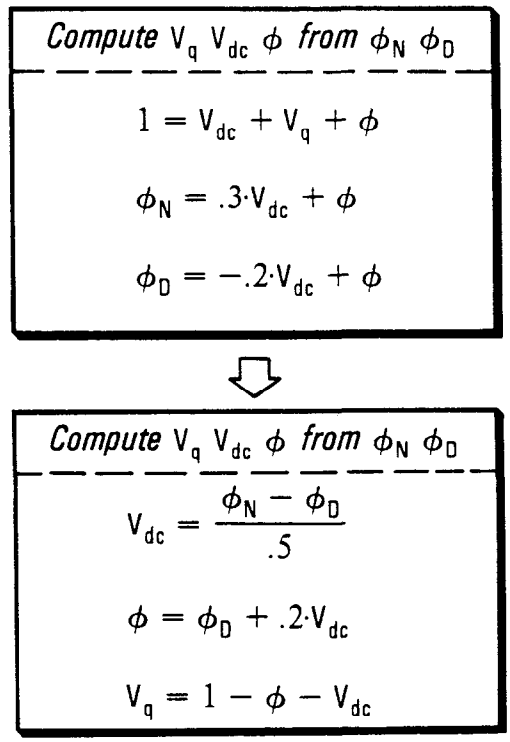


Figure 15. Formal Manipulation: Reformulation within a Formalism.

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 anomolous conditions as special cases which a target program might encounter when it is run on real data

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 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 transformation shown in Figure 14 involves the reformulation of the geometric problem posed earlier into an algebraic one.

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. As illustrated in Figure 15, the problem involving three equations in three unknowns may be reformulated as an equivalent upper-triangular system of equations.

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, the transformation shown in Figure 16 yields a sequence of assignment statements

Implementation Selection The result of the formal manipulation activities is a set of simple algorithms described in terms of general computational concepts dealing with 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.[7] 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* Since most programming languages cannot implement all three abstract data types directly, techniques for representing instances of the data types must be selected. For example, the porosity log measured by the neutron tool is abstractly viewed as a mapping of depths to porosity values. Logs are typically implemented by establishing a correspondence between depths and integers (in effect, a kind of discretization) and then indexing the log by the integers This transformation is illustrated in Figure 17 With several more steps, this representation can be refined into two arrays, one for depths and one for log values.

[7]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

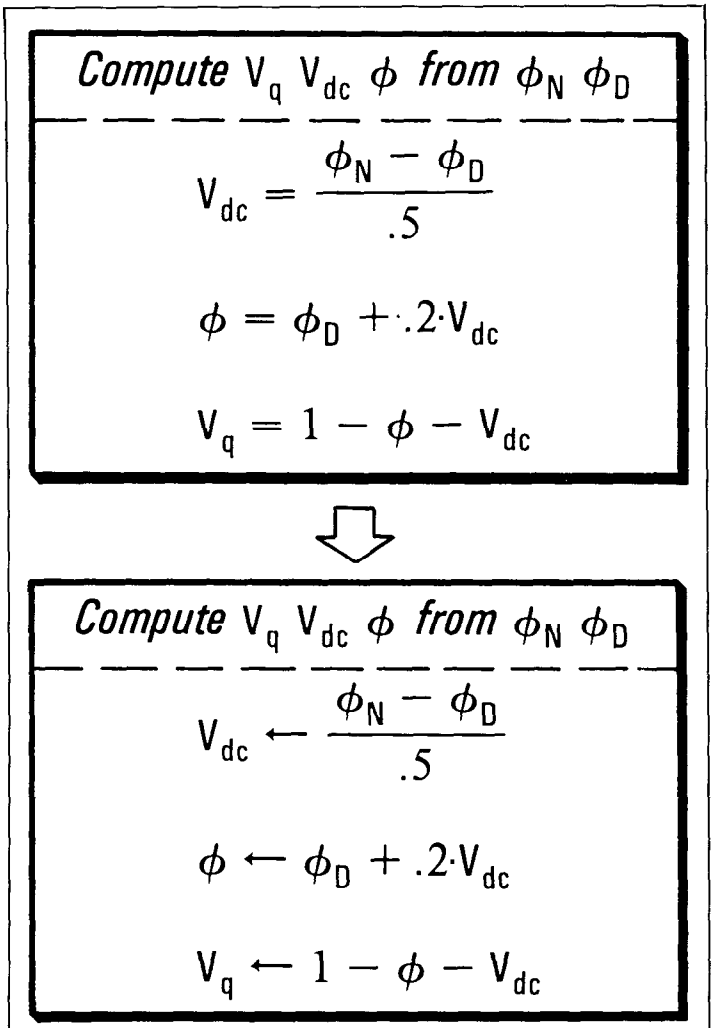


Figure 16. Translation into an Algorithmic Formalism.

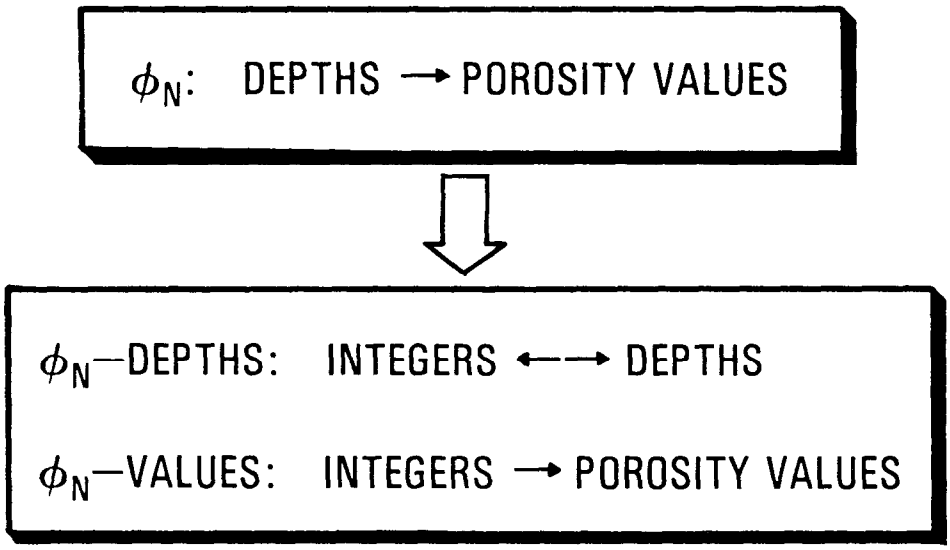


Figure 17. Implementation Selection: Data Type Refinement

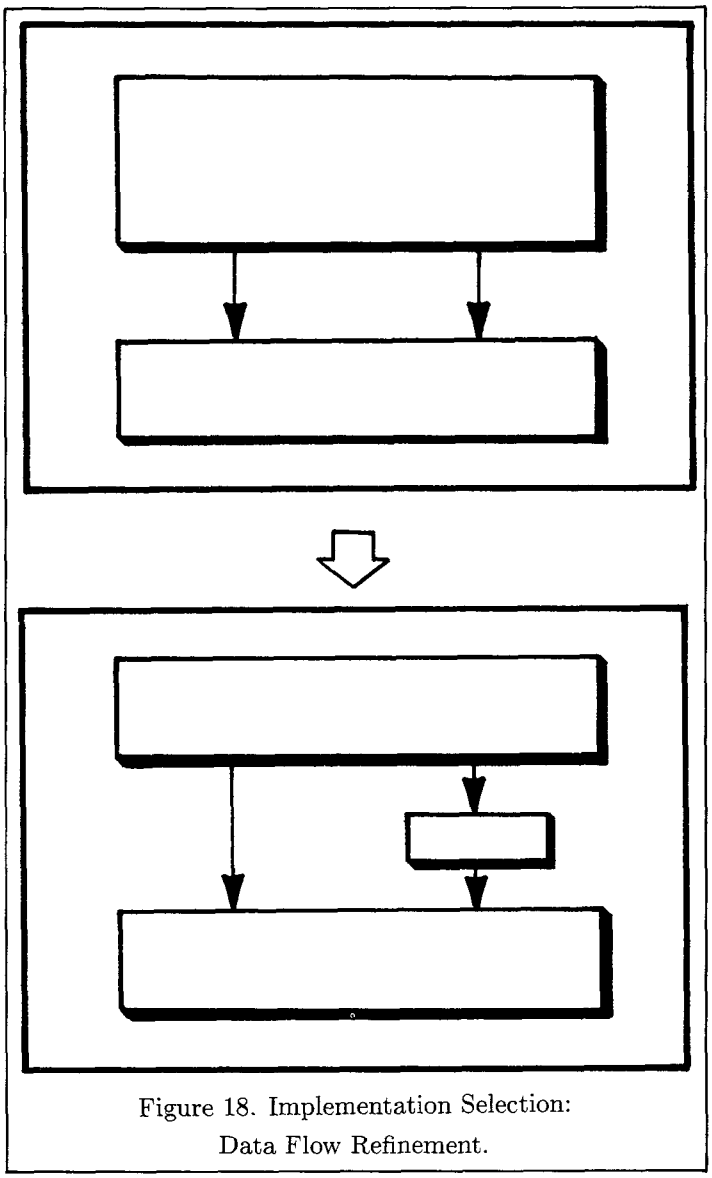


Figure 18. Implementation Selection: Data Flow Refinement.

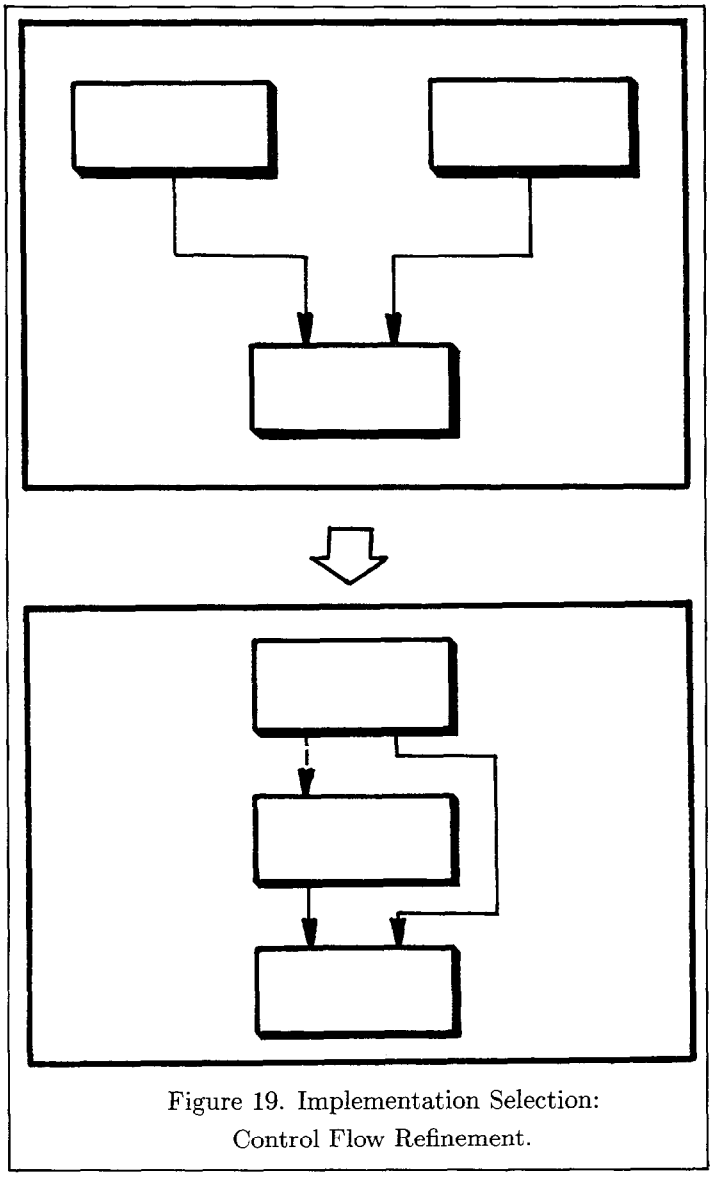


Figure 19. Implementation Selection: Control Flow Refinement.

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. Although many of the data flow decisions seem relatively mundane (*e g.*, whether or not to use a variable) there are also some rather exotic possibilities, such as the conversion of one representation to another in the middle of a computation, as suggested in Figure 18. For example, a set may be represented as a mapping of potential elements to Boolean values while it is being built, and then changed to a sequence to make enumerations easier This is a rather powerful technique — one which both humans and machines could profitably use.

3. *Control flow refinement.* Control flow techniques also differ among target languages. Among the relevant concepts are sequencing, iteration, recursion, conditionality, and function invocation The dashed arrow in Figure 19 illustrates a simple sequencing of computations which are only partially ordered by data flow constraints.

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.

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.

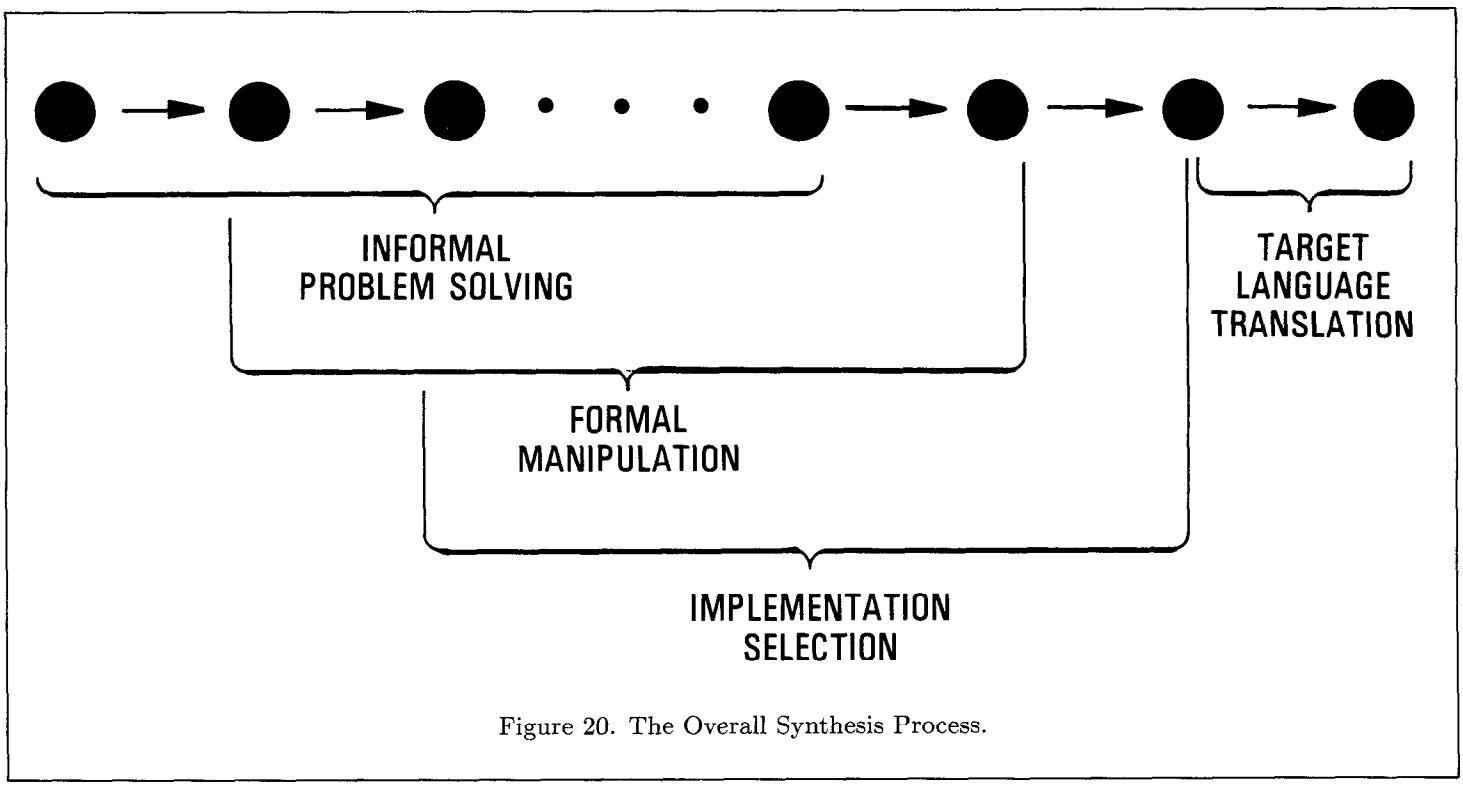


Figure 20. The Overall Synthesis Process.

The Synthesis Process. Since each activity involves the application of transformations, the overall process can be modeled as a sequence of transformations. In other words, the overall process is an instance of the transformational paradigm which has received considerable attention lately (*e.g.*, Balzer 1981, Cheatham 1979, and Kant & Barstow 1981).

Although the activities of the ϕ_{NIX} model were described separately, the overall process cannot be broken into four distinct stages corresponding. As suggested by Figure 20, there is a gradual shift in emphasis from informal problem solving through formal manipulation to implementation selection. To illustrate this mixing of activities, consider the use of an optimization strategy to deal with a complex system of equations. 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. (See Figure 21.) 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.

Characterizing the Activities. We may use this space to characterize the different activities:

- *Informal problem solving.* Due to the informal nature of informal problem solving, the transformations may not preserve equivalence. As a simple example, different elaborations of the consensus concept would lead to programs which are not equivalent, but which might all be plausible from the point of view of a log interpreter Another aspect of informal problem solving is that there are likely to be many alternatives, of which only a small number don't lead to dead-ends Thus, the primary concern will be to avoid dead-ends — almost any completable path is acceptable. This suggests a highly exploratory and opportunistic strategy.
- *Formal manipulation.* Formal manipulation seems to be much more cleanly structured There are probably only a few transformations which produce algorithms directly; the other transformations are used to satisfy preconditions on these. For example, of the three transformations given earlier, the first satisfies preconditions of the second, which satisfies preconditions of the third, which produces an algorithm. Thus, simple backward-chaining is probably the right strategy.
- *Implementation selection.* During implementation selection, most paths will complete successfully and more traditional search strategies are probably appropriate, including quantitative evaluation functions and explicit construction of much of the space.

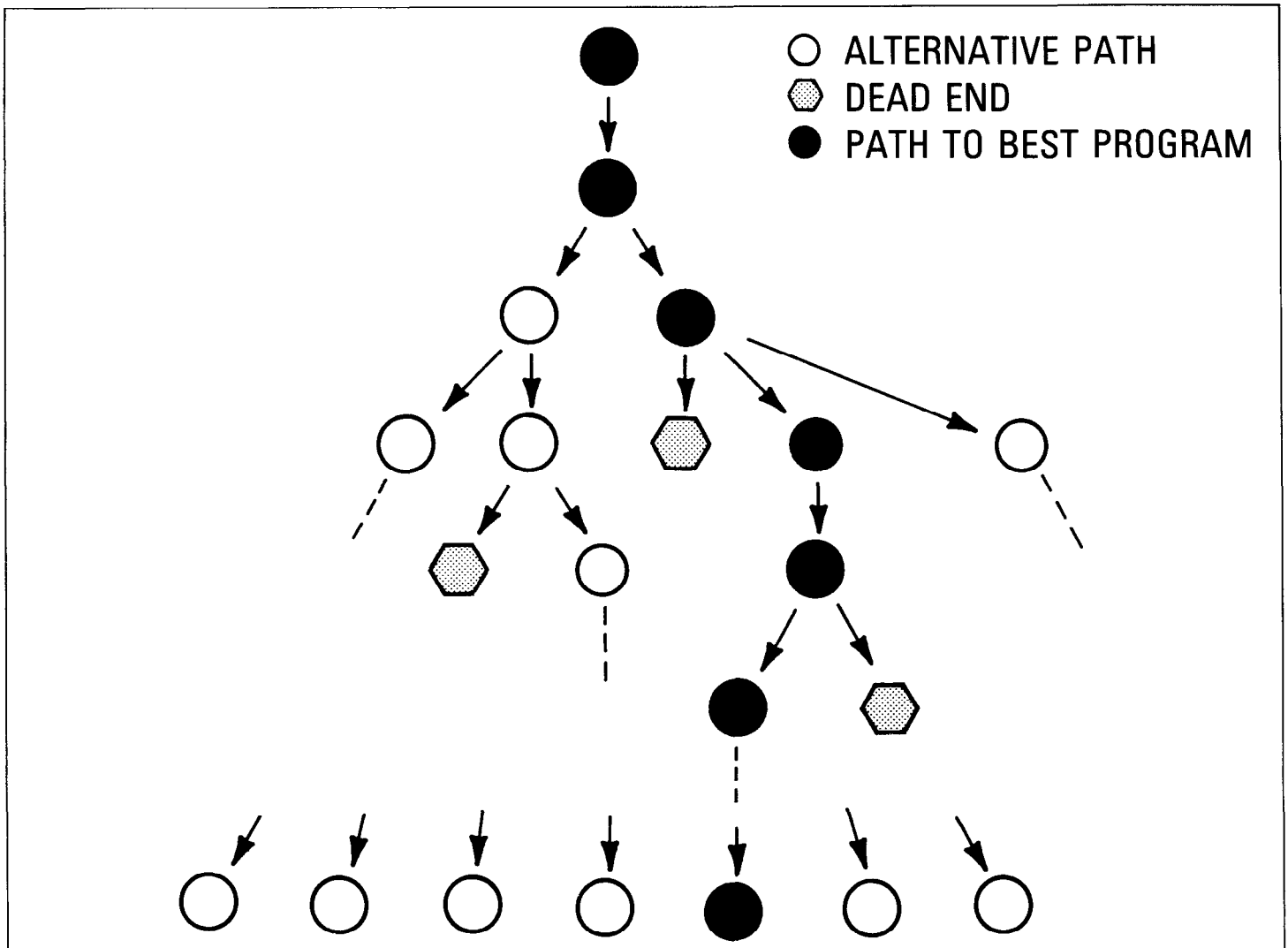


Figure 21. Search Space Defined by Transformations.

The refinement paradigm as developed in PSI's synthesis phase (Kant & Barstow 1981), seems to fit well here.

- *Target language translation.* This activity is quite straightforward, involvng little or no search.

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.[8] Figure 22 shows the roles played by various types of knowledge during the different 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 (*e g* , the end points of the linear GR-S_{wb} relationship). They also play a selective role during formal manipulation and implementation selection by providing the basic information needed to make certain decisions (*e.g* , the plausible range of a physical value).
  - —*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 (*e.g.*, the expression of the linear GR-S_{wb} relationship)
  - —*Problem-solving heuristics:* These are used during informal problem solving to help reduce problems to simpler problems; in effect, they play both a tranformational and a selective role (*e g* , the reduction of the original bound water problem into three simpler ones)

[8] In terms of traditional search theory, knowledge is embodied both in operators and in heuristics

	I.P.S.	F.M.	I.S.	T.L.T.
LOG INTERPRETATION				
Facts and Relationships	T	S	S	
Formulation Mechanisms	T→			
P.S. Heuristics	T/S			
MATHEMATICAL FORMALISMS				
Taxonomy of Concepts	T			
Analytic Mechanisms	S			
Reformulation Mechanisms		T		
Algorithmic Transformations		T→		
PROGRAMMING TECHNIQUES				
Taxonomy of Techniques	S	S	T→	
Efficiency Analysis	S	S	S	
TARGET LANGUAGE				
Taxonomy of concepts	S	S	S	
Surface Syntax				T→

T — TRANSFORMED ROLE
S — SELECTIVE ROLE
→— INTRODUCES CONCEPTS FOR NEXT ACTIVITY

Figure 22. Characterizing the Knowledge.

- Mathematical formalisms (simple statistics, analytic geometry, algebra)
  - — *Taxonomy of concepts:* This plays a tranformational role during informal problem solving by indicating specializations for general concepts (*e.g.*, the use of the arithmetic mean as a consensus technique)
  - — *Analytic mechanisms:* These play a selective role during informal problem solving (*e.g* , the identification of the implausible regions of the ϕ_N-ϕ_D crossplot).
  - — *Mechanisms for reformulation within a formalism or into a different formalism:* These play a tranformational role during formal manipulation by transforming problems into more convenient forms (*e g* , the reformulation of the geometric problem into an algebraic one, and the solution of the system of equations)
  - — *Mechanisms for translating a formal problem statement into an algorithmic form:* These play a transformational role during formal manipulation by producing algorithmic modules (*e g.*, the translation of the solved system of equations into a sequence of assignment statements).
- Programming techniques
  - — *Taxonomy of programming techniques (data flow, control flow, data types):* This plays a tranformational role during implementation selection, since it embodies certain refinement relationships (*e g.*, knowledge of alternative representations for mappings) This also plays a selective role during for-

mal manipulation, since it provides the target set of concepts for the activity

— *Mechanisms for analyzing the efficiency of specific techniques:* These play a selective role during implementation selection. (*E g.,* knowledge of the efficiency characteristics of alternative mapping representations). It also plays a selective role during informal problem solving in that certain transformations might be rejected on efficiency grounds.

- 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 For example, FORTRAN does not establish a new context for each subroutine call, so the use of recursive subroutines would be filtered out.

— *Mechanism for translating into the syntax of the language:* As the essence of the target language translation activity, this plays a transformational role.

In looking at the overall picture, note that domain knowledge is used during each of the activities, but in different ways. During informal problem solving, it is the source of many of the transformations. During later activities, it is used primarily to select from among alternative transformations. Note also that the domain knowledge is not specifically related to the programming task, it is simply used by it.

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.

References

Archie, G. (1942) The electrical resistivity log as an aid in determining some reservoir characteristics. *Petroleum Technology,* 5:1, January.

Balzer, R. Transformational implementation: An example (1981) *IEEE Transactions on Software Engineering,* 7:1, January

Balzer, R , Goldman, N., & Wile, D. (1977) Informality in program specification. *IJCAI 5,* Cambridge, Massachusetts, August.

Barstow, D (1979) *Knowledge-based Program Construction.* Elsevier-North Holland, New York

Barstow, D. (1982) The roles of knowledge and deduction in algorithm design In J. E. Hayes, D Michie, Y -H. Pao (Eds), *Machine Intelligence 10,* Ellis Horwood Limited, Chichester

Barstow, D., Duffey, R , Smoliar, S., & Vestal, S. (1982) An automatic programming system to support an experimental science. *Sixth International Conference on Software Engineering,* Tokyo, Japan, September.

Barstow, D., Duffey, R , Smoliar, S , & Vestal, S. (1982) An overview of ϕ_{NIX}. *AAAI-82,* Pittsburgh, Pennsylvania, August, 367-369.

Bibel, W (1980) Syntax-directed, semantics-supported program synthesis. *Artificial Intelligence,* 12:3, October.

Bogen, R. *et al,* (1975) MACSYMA *Reference Manual.* Massachusetts Institute of Technology, Laboratory for Computer Science

Cheatham, T., Townley, J , & Holloway, G. (1979) A system for program refinement. *Fourth International Conference on Software Engineering,* Munich, Germany, September. Reprinted in D. Barstow, H. Shrobe, E. Sandewall (Eds.), *Interactive Programming Environments,* McGraw-Hill, New York, 1984

Duffey, R., & Smoliar, S. (1983) From geological knowledge to computational relationships: A case study of the expertise of programming *Workshop on Program Transformation and Programming Environments,* September

Floyd, R (1971) *Toward interactive design of correct programs* Stanford University, Computer Science Department, CS-235.

Green, C. (1969) Application of theorem proving to problem-solving. *IJCAI-X,* Washington, D.C., May

Green, C (1977) A summary of the PSI program synthesis system. *Fifth International Conference on Artificial Intelligence,* Cambridge, Massachusetts, August.

Green, C , & Barstow, D. (1977) A hypothetical dialogue exhibiting a knowledge base for a program understanding system In E. Elcock & D. Michie (Eds.), *Machine Intelligence 8,* Ellis Horwood Limited, Chichester.

Kant, E. (1981) *Efficiency in Program Synthesis,* UMI Research Press, Ann Arbor

Kant, E. & Barstow, D. (1981) The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis. *IEEE Transactions on Software Engineering,* 7:5, September Reprinted in D Barstow, H Shrobe, & E. Sandewall (Eds.), *Interactive Programming Environments,* McGraw-Hill, New York, 1984.

Katz, S., & Zimmerman, R. (1981) An advisory system for developing data representations. *Seventh International Conference on Artificial Intelligence,* Vancouver, British Columbia, August, 1030-1032.

Low, J. (1978) Automatic data structure selection: an example and overview. *Communications of the ACM,* 21:5, May

Manna, Z., & Waldinger, R (1980) A deductive approach to program synthesis *ACM Transactions on Programming Languages and Systems,* 2:1, January.

Manna, Z , & Waldinger, R. (1975) Knowledge and reasoning in program synthesis. *Artificial Intelligence,* 6:2, Summer

PROSE General Information Manual, (1979) PROSE, Inc , Palos Verdes Estates, California

Rich, C (1981) Inspection methods in programming *Massachusetts Institute of Technology:* AI-TR-604, June

Rich, C., Shrobe, H (1978) Initial Report on the Programmer's Apprentice. *IEEE Transactions on Software Engineering,* 4:6, November. Reprinted in D Barstow, H Shrobe, & E. Sandewall (Eds.), *Interactive Programming Environments,* McGraw-Hill, New York, 1984.

Simon, H (1963) Experiments with a heuristic compiler. *Journal of the Association for Computing Machinery,* 10:4, October. Reprinted in H. Simon, & L. Siklossy (Eds.), *Representation and Meaning,* Prentice-Hall, Englewood Cliffs, 1972.

Smith, D. (1983) A problem reduction approach to program synthesis. *Eighth International Joint Conference on Artificial Intelligence,* Karlsruhe, Germany, August, 32-36.

Smith, D. (1980) A survey of the synthesis of LISP programs from examples *International Workshop on Program Construction,* Bonas, France, September.

Swartout, S. & Barstow, D. (1983) Who needs languages and why do they need them? or no matter how high the level it's still programming. *Symposium on Programming Language Issues in Software Systems,* San Francisco, California, June.

Swartout, W. & Balzer, R. (1982) An inevitable intertwining of specification and implementation, *Communications of the ACM,* 25:7, July.

Waldinger, R & Lee, R (1969) PROW: a step toward automatic programming *IJCAI* Washington, D.C.

Waters, R. (1982) The Programmer's Apprentice: Knowledge-base program editing. *IEEE Transactions on Software Engineering,* 8:1, January Reprinted in D. Barstow, H. Shrobe & E. Sandewall (Eds.), *Interactive Programming Environments,* McGraw-Hill, New York, 1984.