LBL-24967 c.2

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

EARTH SCIENCES DIVISION

RECEIVED
LAWRENCE
BERKELEY LABORATORY

SEP 27 1989

LIBRARY AND
DOCUMENTS SECTION

The Applicability of Expert Systems to Risk Analysis for Waste Disposal Problems

D. Billaux, M. Uszynski, and S.M. Benson

March 1988

TWO-WEEK LOAN COPY
This is a Library Circulating Copy
which may be borrowed for two weeks.

LBL-24967 c.2

Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098

DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

LBL-24967

The Applicability of Expert Systems to Risk Analysis for Waste Disposal Problems

D. Billaux, M. Uszynski and S. M. Benson

Earth Sciences Division
Lawrence Berkeley Laboratory
1 Cyclotron Road
Berkeley, California 94720

March 1988

This work was supported by the Repository and Technology Program of the Office of Civilian Radioactive Waste Management of the U. S. Department of Energy under Contract No. DE-AC03-76SF00098.

Abstract

Expert-systems are a new type of computer programs in which knowledge is explicitly represented. An expert system has three main parts: (1) a database for storing axioms and rules of inference called the knowledge-base; (2) an algorithm for constructing proofs, called the inference engine; and (3) a user interface. The content of the system data base at any time can be separated into facts, which depend on the case considered; rules of inferences; and relations between facts. The way inferences and relations between facts interplay in the data base defines the various representations of knowledge which can be used. The possibilities and limits of expert systems are mainly due to the way they treat knowledge. They are much easier to maintain, more accessible to people, and can document the conclusions they reach. But an expert system cannot create new knowledge, and because of the "logical-like" representation of knowledge it cannot reproduce easily common sense or analogous reasoning.

In the field of nuclear waste disposal, expert systems can be of help during the study of a site for licensing, or during the operation of the transport, conditioning, and underground storage activities. During the study of a site, expert systems can help planning the field investigations, interpreting results from these investigations, or performing risk assessment studies. During the operation of the storage plant, expert systems can help for industrial management, management of individual waste canisters from power plant to the underground storage, management of monitoring and warning systems.

WES, (Well-test analysis Expert System) is a prototype built to show what can be achieved using expert systems in a crucial area of the nuclear waste storage program. The inputs to WES are principally the results of pumping tests. The output consists of both a conceptual model of the underground flow system, and the values of the parameters needed to properly characterize this particular type of flow system. Contrary to classical curve-fitting softwares, WES achieves its goal not by sophisticated numerical algorithms, but by trying to mimic the behavior of an expert well-test data analyst. It uses a highly structured data base, and is able to generate and manipulate hypothetical conclusions.

Table of Contents

List of Figures vii

1.0 INTRODUCTION 1

2.0 EXPERT SYSTEMS 3

2.1 What is an Expert System? 3

2.1.1 The Knowledge Base 4
2.1.2 Constructing Proofs 10
2.1.3 Interfacing with the User 11
2.1.4 Possibilities and Limits of Expert Systems 13

2.2 Examples of Expert Systems 14

2.2.1 The R1/XCON System 14
2.2.2 The Serum Protein Diagnostic Program 15

2.3 Development of an Expert System 18
2.4 Criteria for Selecting Problems 19
2.5 Application to Nuclear Waste Disposal 20

3.0 WES, THE WELL TEST ANALYSIS EXPERT SYSTEM 23

3.1 General Description 23
3.2 Data Structure and Organization 25

3.2.1 Data Structure 25
3.2.2 Hypothetical Worlds 28

3.3 Rule Architecture and Runtime Behavior 29

3.3.1 Data Extraction 29
3.3.2 Graphics 31
3.3.3 Data Filtering 31
3.3.4 Pressure Derivative 33
3.3.5 Aggregated Description of Curves 33
3.3.6 Generation of Possible Models 34
3.3.7 Selection of One or More Models 34

3.4 Numerical Algorithms 35

3.4.1 Derivatives 35
3.4.2 Straight Lines 36
3.4.3 Humps 36

3.5 Implementation: the ART Programming Tool 39

3.6 Possible Extensions 40

3.6.1 Extending the Analysis of the Present Problem 40
3.6.2 Extending to a Larger Problem 40

4.0 REFERENCES 43

APPENDIX - Listing of the Program 45

List of Figures

Figure 2-1.	Simplest possible data organization: facts, with no structure.	6
Figure 2-2.	Hierarchized facts.	7
Figure 2-3.	Frames.	8
Figure 2-4.	Examples of gauges using active values, from the ART manual (Clayton, 1986).	9
Figure 2-5.	Basic concepts for chaining and search, after Harmon and King (1985).	12
Figure 2-6.	Instrument output from a scanning densitometer, after Weiss and Kulikowski (1984), adapted by Harmon and King (1985).	17
Figure 3-1.	Object architecture for wells in WES.	26
Figure 3-2.	Execution steps.	30
Figure 3-3.	Example screen during a run of WES.	32
Figure 3.4.	Notations for computing derivatives.	37

1.0 INTRODUCTION

Expert systems are software tools which can handle problems where no algorithmic solution can be defined, or where algorithms perform poorly due to, for example, a high number of possibilities ("combinatorial explosion"). These techniques have been used successfully in various fields, such as in medical diagnosis, or molecule definition in organic chemistry by searching for physically possible configurations of a molecule given its global chemical formula. Also expert systems are being used in the interpretation of geochemical surveys by codifying the knowledge of human experts into a set of rules from which the software will be able to make the same inferences these people do, thus putting some of their knowledge at the disposal of many others.

One essential feature of expert systems is the fact that the set of rules used for a given application, i.e. the knowledge in that field, is separated from the mechanism which chooses what rules apply to a given data set, i.e. the inference mechanism. Thus, the set of rules is much easier to modify or expand than the underlying assumptions in any classical program. One first fixes an architecture by choosing the type of representation of knowledge and facts, the inference mechanism and the formalism for the rules which are best suited to the particular problem. One can then build the set of rules gradually, beginning with a subset of the general problem, and increasing the scope of the system gradually until it encompasses the whole field for which it was made.

Expert systems can help solve some of the problems that are or will be posed in the nuclear waste disposal effort, during the study of a site for its licensing, or during the operation of the transport, conditioning, and burying activities. On a given site, the predesign of the repository may be helped by a design-support expert system, and the preliminary characterization studies may be optimized by a planning expert system. During the detailed study of a site, expert systems can be helpful on many different topics. First they can help planning the

field investigations. Second they can be used for interpretation of field data: geochemistry, water tests, geophysics. Finally they can be very helpful for risk assessment studies. Scenario generation and analysis of consequences for example are two areas in which expert-systems can help. During scenario generation, an expert-system can be used to efficiently discard unwanted scenarios as early as possible. Also, generating scenarios requires the generation of many interrelated parameters. Checking the consistency of the whole set of parameters together with the soudness of the scenario can be done by a design expert-system. During the operation of the storage plant, expert systems can help for industrial management, management of individual waste canisters from power plant to burying, management of monitoring and warning systems.

This report investigates the specificity of expert system tools in view of their application to nuclear waste disposal and related activities. The nature of expert systems and their unique characteristics are first discussed, and several examples are given. The trial and error process used for building them is then outlined. The type of problems they can solve is emphasized. We have chosen one problem, well test interpretation, as an example and begun the development of a prototype system. Upon completion, this program will interpret the pressure versus time curve obtained during a pumping test in order to draw conclusions about the type of hydrologic regime underground and provide an estimate of the relevant hydraulic parameters. The current state of this effort is described, and a listing of the program in LISP and ART languages is given in the appendix.

2.0 EXPERT SYSTEMS

A general overview of expert systems is presented below. Numerous text books are available on this subject, and some are cited in the reference list if a more in-depth description is needed.

2.1 What is an Expert System?

An expert system (Bobrow et al., 1986) is a computer program that applies substantial knowledge of specific areas of expertise to the problem-solving process. It is characterized by narrow specialization and competence. The main difference between an expert system and a classical computer program is that an expert system uses technology for the explicit representation of knowledge. In fact, these programs are often called knowledge based systems.

How much competence must such a system show in order to deserve the "expert" qualification? There is no clear-cut answer to this question. Acting like an expert means producing high quality results in a minimal time, by taking advantage of tricks of the trade. In particular, rules of thumb generally enable the expert to jump quickly to a small number of reasonable hypotheses, instead of searching blindly through large numbers of hypotheses. Also, expertise can be judged by how it degrades when approaching the boundary of its domain, what is called the robustness of its knowledge. An expert does not fail altogether when confronted with a problem slightly outside his domain, but his performance degrades smoothly.

An expert system has three main parts (Denning, 1985): (1) a database for storing axioms and rules of inference called the knowledge-base; (2) an algorithm for constructing proofs, called the inference engine; (3) a user interface.

2.1.1 The Knowledge Base

The knowledge contained in the data base can be of two types, shallow and deep. Shallow systems do not resort to general principles in order to draw conclusions. They generally arrive to their conclusion in a small number of steps. On the other end, deep systems derive their conclusions from knowledge of a model of the phenomena in their domain. Their proofs will tend to be longer, but they will be more likely to exhibit robustness in the sense explained above. Shallow systems are faster because they possess a knowledge that is already fine-tuned to the problems they are supposed to tackle. The tradeoff is more rigidity, since the fine-tuning is relevant only for a narrow area. Deep systems are more robust because they contain a knowledge which is expressed in more general terms. When designing an expert system, a balance must be kept between these two extremes, in order to achieve sufficient speed together with some flexibility.

The content of the system data base at any time can be separated into factual knowledge and inferential and relational knowledge.

Factual knowledge is basically a statement of facts, and depends on the case considered, i.e. "there is a hump of height x on the curve". A simple way of representing such facts is the Object-Attribute-Value triplet. In our example, the triplet would be Object: "hump", Attribute: "height", value: "x".

Inferential knowledge states how conclusions can be reached from the facts in the data base, i.e. "a hump in the beginning may mean there is well-bore storage". This is often represented by rules. A rule is simply constituted of a left-hand-side in which a certain number of conditions are enumerated, and a right hand side in which some actions are stated. The left-hand-side conditions are called premises, and the actions on the right-hand-side generally consist of performing changes in the data base, most times in the factual part of the data base. Contrary to the factual knowledge, the inferential knowledge is generally static: it does not depend upon the case under study, and is always present in the database.

Relational knowledge represents what relationships there are between the facts in the data base. For example "a hump is one of several possible features of a well-test curve" links the fact

''hump'' and the fact ''well-test curve''. It may either be incorporated by structuring the facts in the data base, or be partially translated into inference knowledge. Structuring the facts in the data base results in a simpler set of rules, and when large amounts of data are processed it makes the facts more tractable. But more computational overhead is put on the computer. Both the inferential knowledge and the relational knowledge are the actual formalization of the human expert's knowledge. The way inferences and relations between facts interplay in the data base defines the various representations of knowledge which can be used.

If the relational knowledge is not used directly, the factual part of the data base is merely a collection of facts stated independently of each other (Figure 2-1). All the knowledge of the expert is stated in the shape of rules. This means that for example similar facts which represent a characteristic common to a whole group of objects must be stated for each object.

The relations between facts may be used to structure the data base, with all the inferences still based on rules. This time, a characteristic is assessed only once for a class of objects, and these objects are assessed as being part of the class. Any fact is stored only at the relevant degree of generality, and is inherited by all objects spawned below it (Figure 2-2). The facts pertaining to a given object are called its ''slots''.

In the description of objects at various levels of generality described above, all the inferential knowledge is still written in the shape of rules. In the "frame" representation of knowledge however, part of the inferences are made directly by the objects. This is achieved by defining objects as collections of slots as above, and by allowing any given slot of an object to be attached to facts or procedures, or both (Figure 2-3). Then for example, when a value is needed from a slot attached to a procedure, the procedure is executed to return the needed value. Or when the value of a slot containing a fact is changed, it may trigger a procedure used for instance to update a graphical representation (gauge or dial, etc.., Figure 2-4). This particular slot is then called an ''active value''. If this kind of programming is extended to include most or all of the inferential knowledge in the object structure, then the programming style obtained is called ''object-oriented programming''.

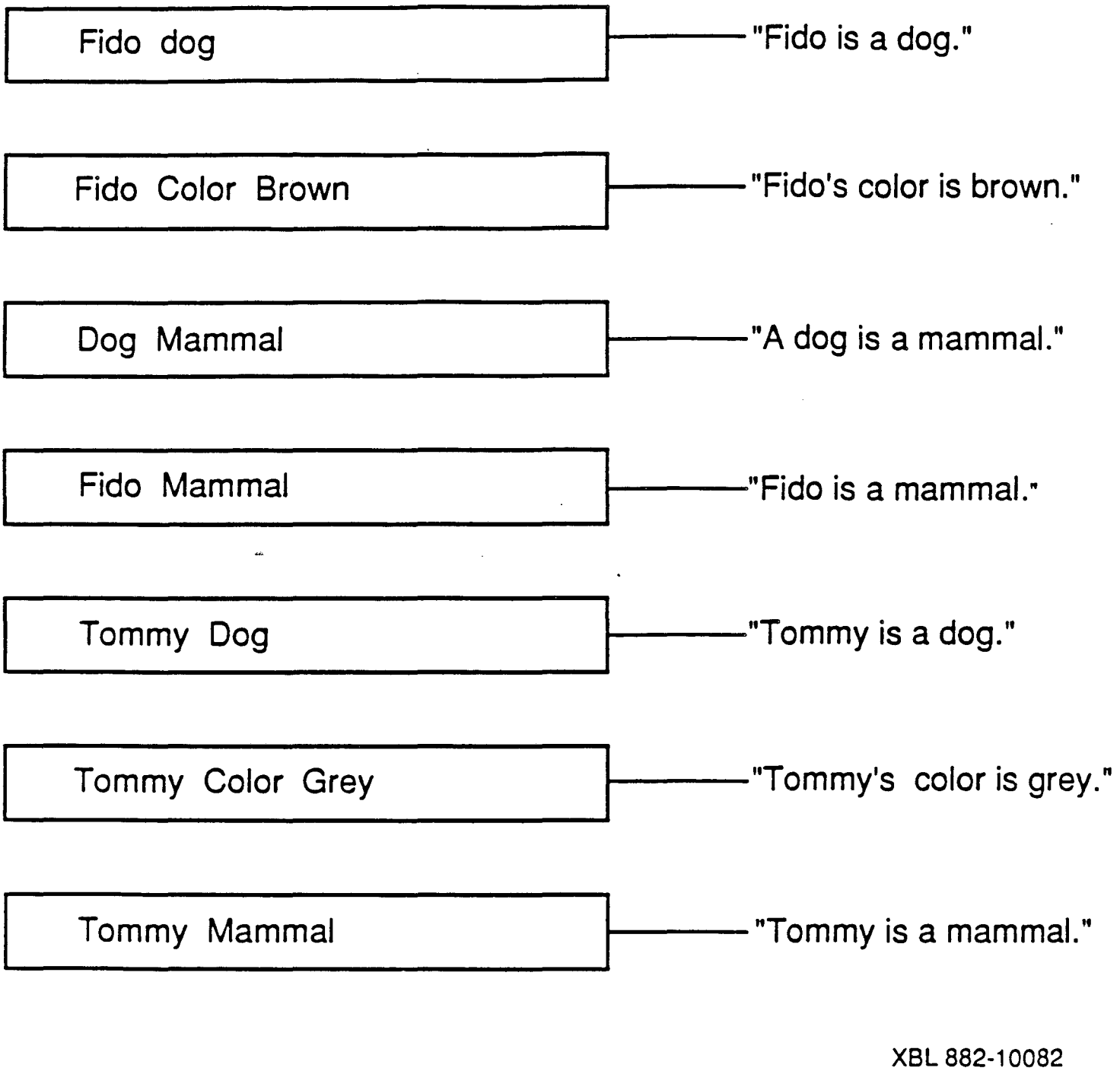
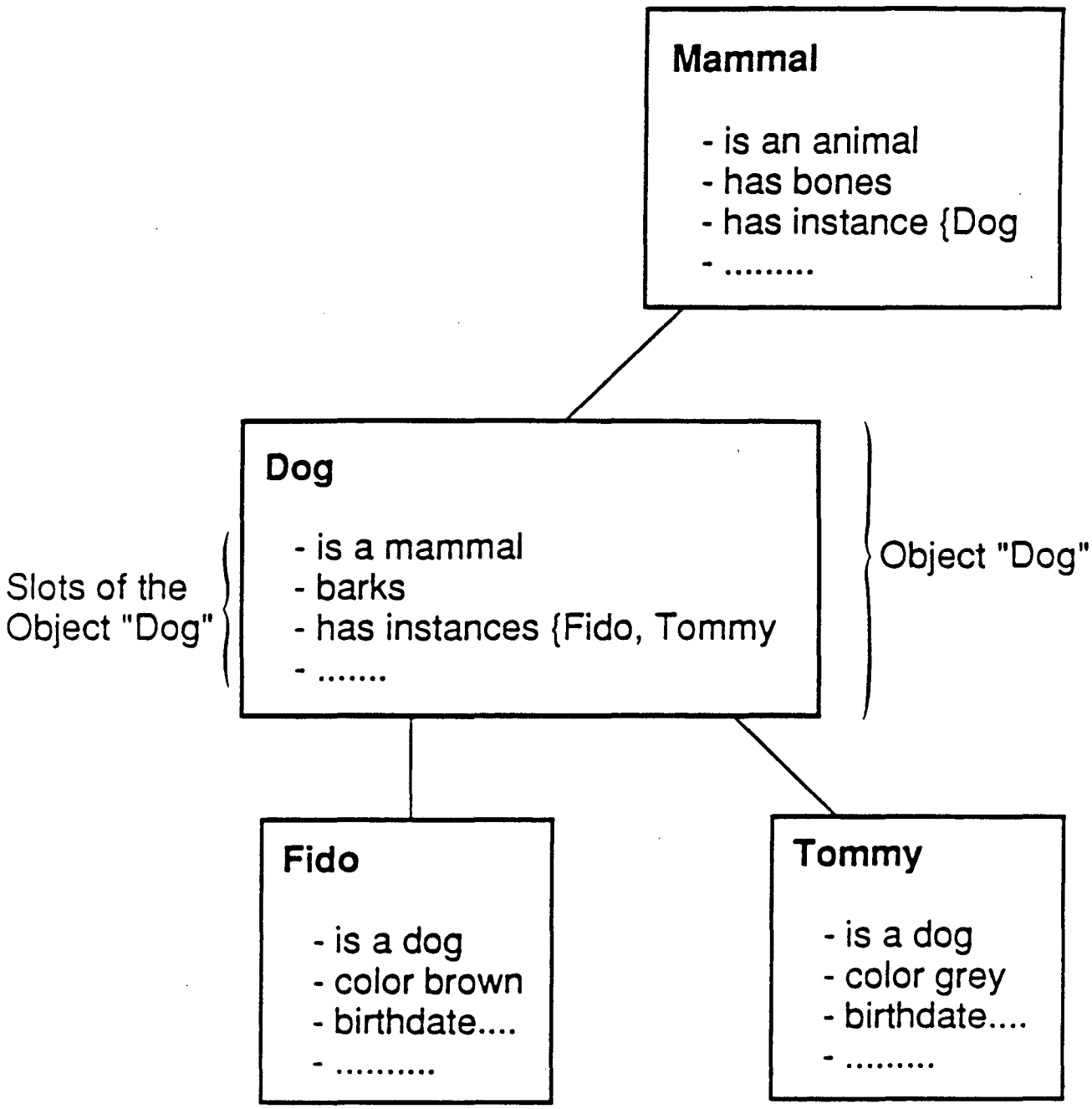


Figure 2-1. Simplest possible data organization: facts, with no structure.



The facts that fido "is a mammal", "barks", "is an animal", "has bones", do not have to be stored, since they are inherited from objects higher in the hierarchy.

XBL 882-10083

Figure 2-2. Hierarchized facts.

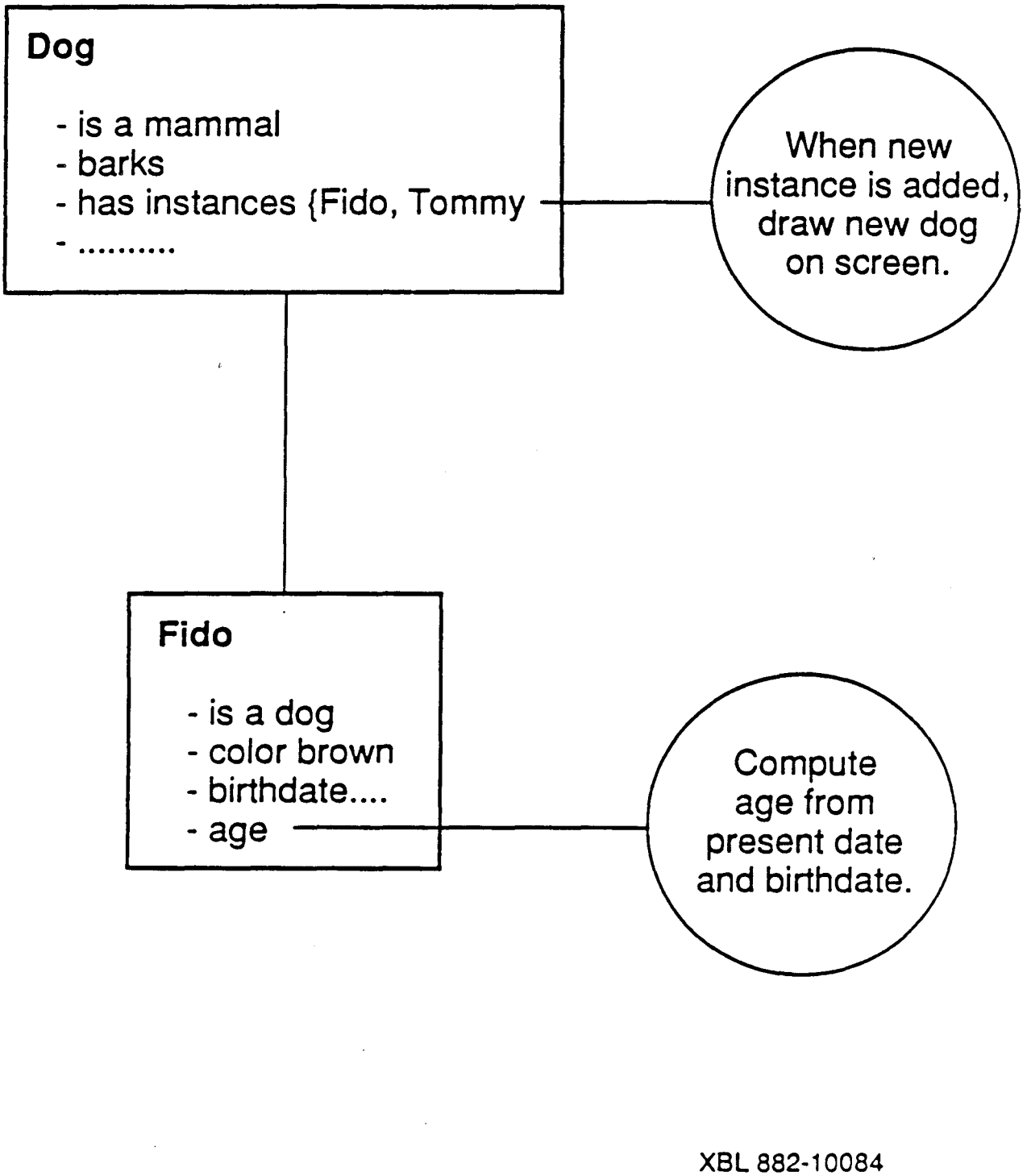


Figure 2-3. Frames.

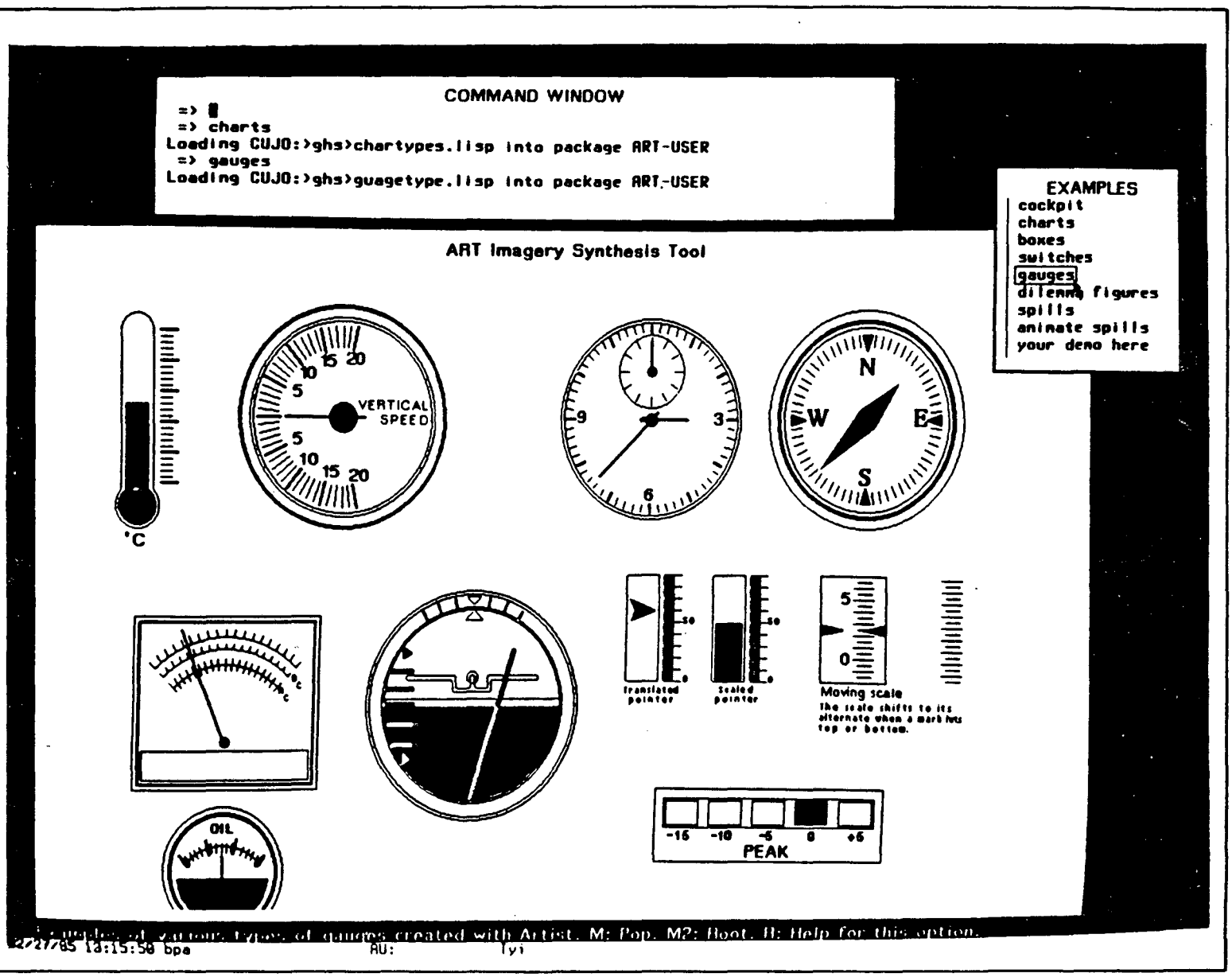


Figure 2-4. Examples of gauges using active values, from the ART manual (Clayton, 1986).

2.1.2 Constructing Proofs

In order for a system to reason, it must be able to infer new facts from the facts already in the data base. Two control mechanisms are commonly used. If **forward chaining** is used, the system looks for the rules whose premises are matched by facts in the data base. Among these rules, one rule is chosen. Then this rule is "fired", which means that the consequence part of the rule is executed and the data base is changed accordingly. Then the system examines the rules again. The process of choosing the rule which will be fired among all rules that have their premises verified is called "conflict resolution". This control mechanism is sometimes also called data driven control.

On the other hand, a **backward chaining** inference engine starts with a goal and tries to find which conditions must be verified for this goal to be reached. A goal here is simply a given state of part of the data base. It can be just the presence of a given fact or of a collection of facts, i.e. "There is well-bore storage." Backward chaining thus involves scanning the rules and selecting the ones whose right-hand-side actions achieve the goal. If part of their premises are not verified, then these non-verified premises become sub-goals and the process is repeated until either a way to verify the goal is found or a conflict between a sub-goal and the state of the data base is reached, in which case the particular goal is discarded. Note that several different goals can be started with, and which one succeeds may be the outcome of the program. Backward chaining also involves conflict resolution, since at any step of the reasoning process, several rules may have the desired effect. This type of inference control is also called goal directed control.

The difference between forward chaining and backward chaining can be summarized as follows. When using forward chaining, one repeatedly asks the question: "where can this data lead me?" and the paths of reasoning explored are all the changes of the data base from the initial one which are permitted by the body of rules. On the contrary, when using backward chaining, one keeps asking the question: "how can I reach this - or these - particular conclusion(s) from the data base" and the paths of reasoning explored are all the possible states of the data base which would lead to the desired conclusion(s) through changes permitted by the body of rules. Note

that backward and forward chaining are not incompatible. In fact many recent expert systems tend to use both control strategies, in order to take advantage of their complementary qualities. These qualities stem logically from their definitions. If there is not too much data to start with, but there are many possible different conclusions, then a forward chaining control starting from the limited initial data will be efficient. But if there is a lot of data then chaining backward from a few likely hypotheses will be more efficient.

From the discussion above, one can sense that drawing inferences can be simply stated as searching in all the possible states of the data base for one which would satisfy a set of conditions defined both by the initial data entered and the rules for transforming this data. Backward and forward chaining specify in which "direction" the search task is implemented. Once this direction is chosen, the way the search is performed is still open. Two simple strategies are the depth-first search and the breadth-first search.

In a **depth-first search**, a path of reasoning is followed to its ultimate consequences before another path is considered. In a **breadth-first search**, each possible path of reasoning is followed one step at a time, so that all possible paths are searched simultaneously. Thus depth first-search dives deeply into the search graph, whereas breadth-first search descends uniformly across all possibilities. In the same manner as backward and forward chaining can be combined, a search does not have to be purely depth- or breadth-first. In the implementation of an example system described in section 4, the search is generally depth first, but with the possibility of searching several different paths of reasoning at the same time. This amounts to giving some "breadth" to the search. Figure 2-5 summarizes the basic concepts of backward and forward chaining, and of depth- and breadth- first searches.

2.1.3 Interfacing with the User

In many existing expert systems, the part of the program which requires the largest amount of computer memory, and also the longest development effort is not the knowledge base or the inference engine. In fact sometimes even adding up these two parts of the system does not come to the size of the user interface. In this respect, expert systems just continue a general trend in

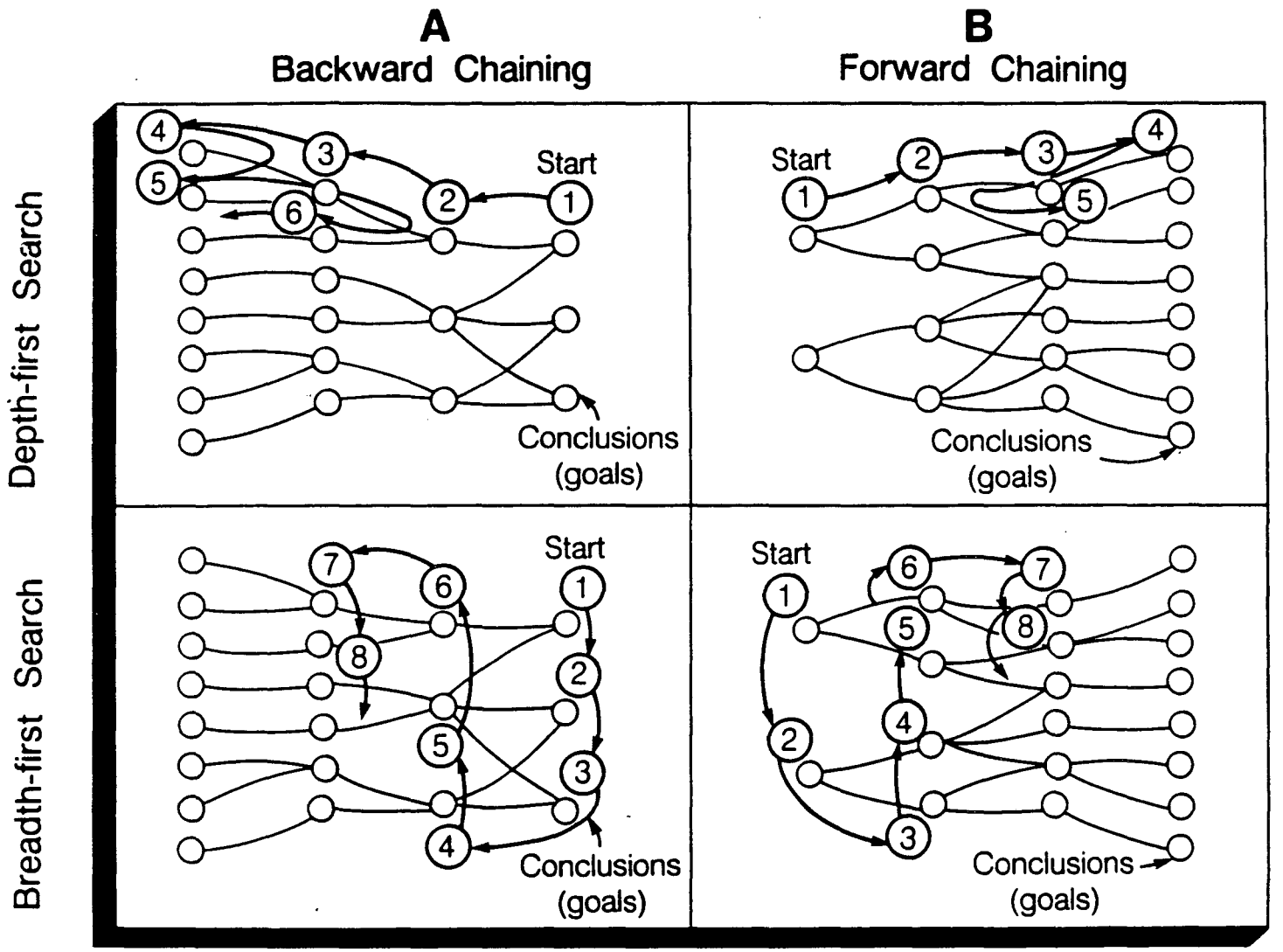


Figure 2-5. Basic concepts for chaining and search, after Harmon and King (1985).

programming towards user-friendliness. Because many view expert systems as endeavors to mimic "human" characteristics with a machine, people expect these programs to conduct dialogue with them in as natural a way as possible. This means representing data and results on the computer screen or a printout in the same fashion as an expert in the field would chose to represent it.

2.1.4 Possibilities and Limits of Expert Systems

The possibilities and limits of expert systems are mainly due to the way they treat knowledge. The explicit representation of knowledge makes these programs much less cryptic than classical ones, and therefore much easier to maintain or modify. The formalization of knowledge makes the expertise available to more people, and also provides the possibility of multi-disciplinary systems. A third important quality of the knowledge representation is its transparency: the decision process for any given run of the program can be integrally documented and traced back.

Present-day expert systems have two limitations which are likely to stay for some time. First, an expert system cannot create new knowledge. Not only do we need to be able to solve a problem, but also we need to know how we solve it, in order to be able to design an expert system for that particular problem. The second limitation is that because of their "logical-like" representation of knowledge expert systems cannot reproduce properly common sense or analogous reasoning.

2.2 Examples of Expert Systems

The two examples described below are very different in terms of size, of number of users, of development time. R1/XCON is widely regarded as one of the most successful systems built yet. It is run on a mainframe computer and used by the Digital Equipment Corporation to configure VAX and PDP-11 computers. The second example is the Serum Protein Diagnostic Program. This system interprets the results of a laboratory test, serum protein electrophoresis. The program was translated into microprocessor assembly language, then this program was put on a chip interfaced with an instrument performing the test. In this way, the instrument provides not only numerical results but also an interpretation consisting of a few sentences. The instrument and the accompanying expert system are now widely disseminated.

2.2.1 The R1/XCON System

The R1 project was started in 1978 by John McDermott at Carnegie-Mellon University and the Digital Equipment Corporation (DEC). The process of building this system has been described extensively by McDermott (McDermott, 1980, 1981) and others (Bachant and McDermott, 1984).

DEC does not market preconfigured systems; instead, it offers a customer a wide selection of components to choose from. In 1979, for example, some 420 components were associated with a VAX-11/780. Thus, most of the systems DEC sells are one-of-a-kind systems (Harmon and King, 1985). The input to the system is the customer order. R1/XCON determines what if any substitutions and additions are needed to make the order consistent and complete. It then designs the layout of the various components and outputs a set of diagrams showing the spatial and logical relationships between the components. These diagrams are used by the technicians who assemble the system.

Three previous attempts by DEC to automate the processing of customers orders for VAX computers had failed. These three earlier attempts had been built using standard programming technology. The major problem had been the difficulty of changing the system as additional

knowledge was gained.

R1/XCON is a rule-based system, where premises test such things as the state of the problem-solving process, availability of components, and connections between the components in the configuration. After a year of work, the prototype contained about 750 rules. It was subjected to a formal evaluation and acceptance test during which it was asked to configure the 50 most recent orders DEC had received at the time. Errors were found in twelve configurations. After the rules responsible for these errors were changed R1/XCON reconfigured correctly the 50 orders. It was at that time considered expert enough to be used routinely. But after the program was installed in the working environment at DEC and began to configure computers, a major effort was needed to bring it to a satisfactory level of expertise. In fact, the program grew to about 3500 rules. These new rules were needed for the following reasons, listed in order of the number of rules added: (1) refine the knowledge to correct errors made by the early system; (2) extend the knowledge to the configuration of another much more complex computer, the PDP 11 line; (3) have the system perform a more complete task (more detailed configuration); and (4) improve output.

One can sense from this case that in the building of a major system, the development of a prototype is hardly the end of the story. In fact R1/XCON still undergoes changes and updates. The maintenance cost of this system is quite high. But given a classical program, maintenance would have simply been impossible.

2.2.2 The Serum Protein Diagnostic Program

By contrast with the large project described above, this code is aimed at solving a much more limited problem. It was built by S. M. Weiss and C. A. Kulikowski, of Rutgers University, and is described in detail in their later book (Weiss and Kulikowski, 1984). This illustrates an interesting approach to gaining acceptance of a small expert system by the end-users. Weiss and Kulikowski analyze the preformance of earlier large scale medical systems, and show that although many of them passed successfully clinical experiences, few are in routine clinical use. This is due to both technical and social factors. One of them is the slow rate of manual

interactive data entry. This can be alleviated when most of the data can be input directly from a medical instrument. Also, they argue that an interpretation program posted between the raw output of a medical instrument and the user is more likely to be accepted because it is seen as an extension of the instrument itself. And the hardware cost is minimal since most modern instruments are already microprocessor controlled, and the microprocessor could be used to host the interpretive program.

Serum protein electrophoresis is performed by a scanning densitometer. The output of the instrument is a curve with several peaks. For each of these peaks, the densitometer computes the area under the curve. A few items (patient identification, age, some features of the curve) are entered manually. From these, the system deduces an interpretive analysis which is printed together with the curve and numerical output and consists of a few lines (Figure 2-6). The different conclusions were ordered carefully and rules were added to suppress the display of redundant statements, i.e. conclusions which are useful in the reasoning but are covered by other statements.

Using a specialized expert system shell, EXPERT, geared towards classification problems, Weiss and Kulikowski were able to produce the system in the course of six months. The final product had 82 rules covering all common cases encountered by a scanning densitometer. The process of model design was the following:

- building of the prototype using EXPERT and a mainframe computer
- testing using several hundred cases
- refinement of the system by the expert
- new cycle of testing and review by a panel of independent experts
- test of the final model on the mainframe
- automatic translation of the EXPERT model to a specialized program and a microprocessor assembly language program
- interfacing of the assembly language model with the instrument. This was done by the

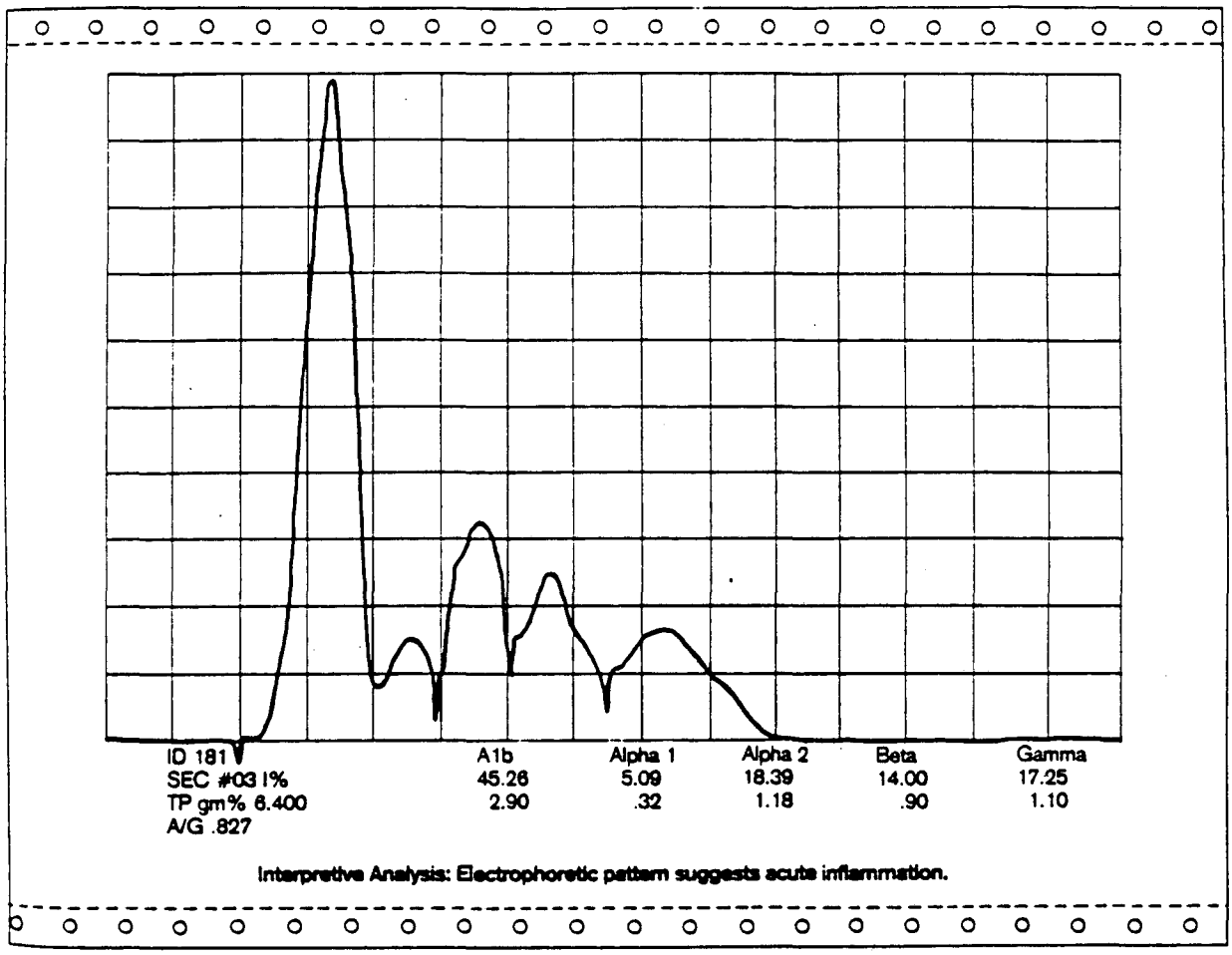


Figure 2-6. Instrument output from a scanning densitometer, after Weiss and Kulikowski adapted by Harmon and King (1985).

company manufacturing the instrument (Helena Laboratories).

The microprocessor version of the system could theoretically have been coded directly in assembly language. However, the extensive testing and revision process the model underwent during its elaboration would have been impossible in practice. The experts testing the model on an expert system shell can use the tools that the shell offers to investigate the logic of the system and understand exactly how the program is making its judgements. Once they are satisfied, the completed knowledge base and a simplified inference engine translated into a lower-level form enable the spreading of the software and its effective use by the community it was intended for.

2.3 Development of an Expert System

As can well be seen from the two examples above, developing an expert system is a trial and error process. This process can be divided into several phases (Hayes-Roth et al., 1983).

(1) Identification of the participants, problem characteristics, resources, and goals. Most expert system attempts involve a single domain expert and a single knowledge engineer at least in the early stages, but many more participants may be involved in the testing procedure. Problem identification includes the definition of main subproblems, the definition of the inputs to the program and of the characteristics of a solution.

(2) Conceptualization. Through repeated interaction between the expert and the knowledge engineer, the key concepts and relations are made explicit, and diagramed. The vocabulary for the field is made explicit. The hierarchy between concepts is studied.

(3) Prototyping. This phase can be divided in two steps. First the problem is formalized, by defining the hypothesis space, the underlying model of the process, and the characteristics and structure of the data. This way partial specifications for the prototype knowledge base are obtained. The second step is the actual implementation of the prototype using a given tool. At this stage, inconsistencies are likely to be found and must be resolved. Also, some experiments on user interface may be started then.

(4) Testing and redefinition. Once the prototype works from start to finish on the first exam-

ples, new examples should be added and the various parts of the system are likely to undergo numerous changes. In fact, earlier versions may be discarded altogether if needed. Building a friendly and natural user interface is critical to the later acceptance of the program. This phase should also include exposing other experts to the program and responding to their views.

2.4 Criteria for Selecting Problems

Due to their qualities and flaws, and also due to the way they are built, expert systems do best in some types of problems, and are likely to perform poorly in some others. A non-exhaustive list of the "good" problems would be (Hayes-Roth et al., 1983) interpretation, or inferring situation descriptions from observables; prediction, or inferring likely consequences of given situations; diagnosis, or inferring system malfunctions from observables; design, or configuring objects under constraints; planning, or designing actions; monitoring, or comparing observations to plan vulnerabilities; debugging, or prescribing remedies for malfunctions; repair, or executing a plan to administer a prescribed remedy; and instruction, or "diagnosing", "debugging" and "repairing" student behavior. R1-XCON is a design system, and the Serum protein electophoresis interpreter is an interpretation system. In fact most existing expert systems fall into one of the three areas of interpretation, diagnosis or design. Other examples of problems developed in some of these fields include the following systems:

- DENDRAL is used to interpret physical measurements on an unknown chemical and build plausible molecular structures for it.
- PROSPECTOR interprets soil and geological deposit data to assess to possibility of finding a mineral resource in a given area.
- CADUCEUS and MYCIN are both medical diagnosis systems.
- PRIDE is an expert system developed at Xerox to assist engineers designing paper transports inside copying machines.

Given a problem of a type listed above, other criteria for chosing an application stem mostly from common sense. The value of solving the problem should be high enough to justify

the cost of software development (from a few person-months for simple systems to several person-years for a very sophisticated one). Test cases should be available in significant number. In fact, trying to build a set of test cases early provides insights about the feasibility of the project. The difficulty of the task should be such that any expert completes it in a range of a few hours of reasoning. A thirty-minutes task is most likely not to be worth the effort, and a two weeks long task is probably too wide in scope, except if it consists in a number of iterations of a shorter task. Critical to the success of an expert system is the choice of a "good" expert. Not only should the expert have mastered his field, but also he should have a strong interest in building the program.

2.5 Application to Nuclear Waste Disposal

The possibilities for the application of expert-systems to nuclear waste disposal have been investigated by conducting interviews with people concerned by the programmation of nuclear waste disposal:

- Edward Patera — Repository Technology and Transportation Division, DOE Chicago Operations
- R. Laughon — Office of Nuclear Waste Isolation Battelle Laboratories, Argonne, Illinois
- Paul Gnirk — RE/SPEC Inc., Rapid City, South Dakota
- Bernard Come — Nuclear Waste Disposal Division, DG XII: D1 - Commission of the European Communities
- Jean Pierre Olivier — Head, Radiation Protection and Waste Management Division, OECD Nuclear Energy Agency

As can be concluded from paragraph 2.4, the contribution of expert systems cannot be the development of one large all-encompassing "Nuclear waste" expert system. In fact, given the present state of the art an attempt to build any general purpose expert-system is very likely to fail. One can more realistically view expert systems as tools of limited scope able to tackle well bounded problems in an efficient and tractable way.

Expert systems can be of help during the study of a site, or during the operation of the transport, conditioning and burying activities.

During the study of a site, expert systems can be helpful on many different topics. First they can help planning field investigation. Second, they can be used for the interpretation of field data, in either geochemistry, hydrology, or geophysics. For example, a code built upon the prototype WES presented below would provide a standardized interpretation of pumping tests in wellbores. Such an interpretation consists of information about the geometry of the underground system parts. In order to predict the behavior of the system, one must use these data in a numerical model. Expert-systems can be used as user interfaces for the complex codes that perform the modeling. Used together with test-interpretation codes, such interfaces would provide a formalized approach to the building of a numerical model from experimental results. Finally, expert-systems can be very helpful for risk assessment studies. Scenario generation and analysis of consequences for example are two areas in which expert-systems can help. During scenario generation, a simple expert-system can be used to difficiently discard unwanted scenarios as early as possible. This would automatically provide documentation on the rationale for discarding any trial scenario. Also, generating scenarios requires the generation of many interrelated parameters. Checking the consistency of the whole set of parameters together with the soundness of the scenario can be done by a design type expert system.

During the operation of the storage plant, expert-systems can help for industrial management, management of individual waste canisters from power plant to burying, and most of all for management of monitoring and warning systems. Such monitoring and warning systems could be fed at their starting point with the results of the modeling and scenario studies performed during the design phase, and assess in real time the difference between the predicted and real behavior.

3.0 WES, THE WELL TEST ANALYSIS EXPERT SYSTEM

WES, (Well-test analysis Expert System) is intended as an example of what can be achieved using expert systems in some crucial areas of the nuclear waste storage program. This program is by no means a finished system. As will be seen below, it currently solves a small part of the different kinds of problems for which it is conceived. However, this prototype should provide a good insight into the complexity of the expert system building task, and into the versatility of the tools that are now available for this purpose.

3.1 General Description

The purpose of this expert system is, in general terms, to produce information about the hydraulic characteristics of the subsurface. Specifically, the output of one run of WES consists of both a conceptual model which describes the nature of the ground water flow system, and an estimate of the parameters needed to properly characterize this flow system. These conclusion are reached through the interpretation of pumping test data. During such tests, the hydrogeologic regime is perturbed by pumping water in or out of a well, and the response of the underground flow system to the perturbation is generally monitored over a long length of time, yielding curves of pressure versus time.

When an expert works on the interpretation of pumping tests, he looks for unique characteristics, or "signatures" on one or several representations of the pressure-time data. Classical representations include semilog and loglog curves, and a newer method uses a pressure derivative curve. Well known types of hydrogeologic systems have different signatures on these curves, such as straight lines, humps, etc. The task of the expert is therefore to draw the curves, recognize these features and draw conclusions from them. An expert is likely to follow several compatible or mutually exclusive hypotheses in his or her search for a conceptual model, until he or she is convinced that one coherent group of hypotheses is more likely than the others.

The guiding principle in the design of WES has been not only to try to reproduce the results obtained by the expert, but also to try to mimic the way the expert reaches a conclusion. The input data are principally the test results, which consist of data points representing the variations of water pressure versus time. The analysis is conducted using the semilog and loglog representations of these input data, and the pressure derivative method (Bourdet et al., 1984). The advantage of this newer method is that the signatures of different types of geologic features are more readily discerned than with conventional analysis methods. One disadvantage is that usually, pressure derivative plots tend to have more random noise than the original data.

In its present state the system can accommodate only a single testing well and a drawdown phase (a constant flow rate following a zero flow period), but it will be extended to handle a buildup phase (zero flow period coming after a constant flow rate period), or more generally a multiple flow rate pumping phase, as well as situations where there is a pumping well and possibly more than one observation well.

Currently, the system possesses knowledge about only a limited set of conceptual models: two general types of reservoirs (homogeneous and double-porosity), a wellbore-storage effect or a fractured well effect, both of which may distort the response in the early part of the data set, and three different boundary configurations: infinite system (no boundary), closed system (no flow boundary) or a constant pressure boundary. Certain assumptions are also implicitly made, such as constant flow rate, and constant atmospheric pressure.

The system is written in ART (Automated Reasoning Tool from Inference Corp., Clayton (1984)). ART provides an easy interface to LISP (LISt Processing language, a widely used artificial intelligence computer language), so part of the system is written in LISP.

The following sections describe respectively the data structure, the architecture of the system, the various algorithms used for computing derivatives or extract global characteristics of curves, some implementation details relative to ART, and propositions for extensions in different directions.

3.2 Data Structure and Organization

3.2.1 Data Structure

The system uses an frame-oriented approach to structure its data and any conclusion or intermediate result reached during execution. The basic architecture of objects refers to wells and characteristics of wells, but objects are also used for graphics to represent windows opened on the screen, curves, mouse-sensitive icons (areas in windows where clicking the mouse has a specified meaning), etc. This use of objects to represent almost any piece of data offers a lot of flexibility and simplifies programming.

The object architecture relative to wells is shown in Figure 3-1. Before execution, only generic objects such as "well" or "well-semilog" are defined, along with all the slots that may be used to characterize these objects. These generic objects are basically empty structures that reproduce our relational knowledge. At execution time, any well-test the program studies (i.e. Lm40) is viewed by the program as a specific instance of the generic object "well". The specific object created by WES (i.e. "Lm40") is called the parent of all the objects describing the knowledge of the system about this particular well-test. WES then creates these other "Lm40-xxx" objects as instances of the other generic objects if needed during the run. For example, "Lm40-semilog", an instance of the generic object "well-semilog", is created if a semilog analysis is performed on well-test Lm40. "Lm40-semilog" is a replica of "well-semilog", but the empty slots in the generic object have been replaced by the specific characteristics of Lm40.

Relations connecting the different objects are created for any specific well-test, i.e. Lm40. These particular facts allow the user or the program to link informations relative to a specific well-test. For example, the relation **semilog** links the parent object "Lm40" to the object containing information about the semilog curve "Lm40-semilog". Each relation has an inverse relation (in this case **semilog-of**). Note that the generic parent object "well" is itself linked by an "is-a" relation to the object "instantiated-window-icon". This relation and this exotically named object are provided by ART and used here to initialize the graphic interface.

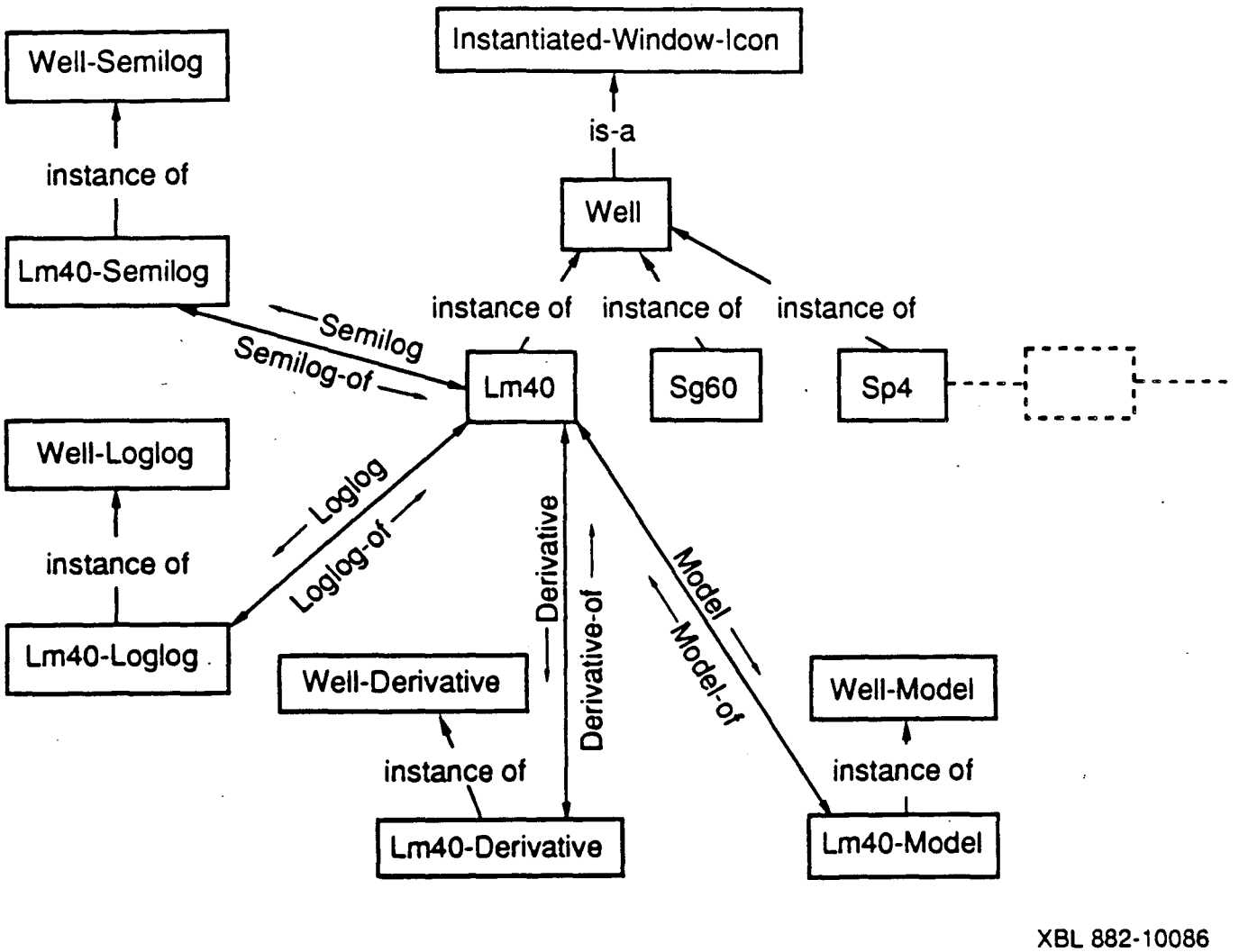


Figure 3-1. Object architecture for wells in WES.

The generic objects may have several different instances simultaneously. Each time a new well is considered for analysis, a new specific parent object is created. For each well-test under study, parts of the generic object structure are duplicated into specific objects when needed for the analysis. Knowledge is kept in these specific objects, therefore avoiding any confusion between the different well-tests.

Specific objects are created from the five different generic objects: well, well-semilog, well-loglog, well-derivative, and well-model. The main slots of these five generic objects are shown in Table 3-1. These slots are always empty, and only the slots of the corresponding specific objects will be filled at run time. Note that a slot may contain either some characteristic, i.e. pressure; a relation pointing to another object, i.e. semilog; or a procedure, i.e. window (standing for "open a window").

Table 3-1: Main slots of the generic objects

Well	Well-semilog	Well-loglog	Well-derivative	Well-model
initial-time initial-pressure time pressure semilog loglog derivative model x-scale y-scale initial-curve common-icons window	semilog-of straight-line semilog-curve semilog-icons	loglog-of straight-line loglog-curve loglog-icons	derivative-of model time p-derivative p-d-derivative straight-line hump derivative-curve derivative-icons	model-of reservoir wellbore-storage fractured boundary early intermediate late

Initial knowledge is stored in the parent object for a given well. This parent object is an instance of the generic object **well**. It consists of the time and pressure data, the dimensions of the display window, and the links to the four other main objects associated with this particular well.

Well-semilog contains knowledge about the semilog curve, such as straight lines. **Well-loglog** contains the same knowledge for the log-log curve. **Well-derivative** contains the pressure

derivative, the second order derivative, and characteristics of the pressure derivative curve, such as straight lines and humps.

Well-model is the object that contains symbolic assertions about the well, that is, deductions reached by the system according to the different characteristics of the semilog, log-log and derivative plots. **Reservoir** describes the overall behavior of the medium. It can take the values *homogeneous* or *double-porosity*. **Boundary** describes the boundary behavior of the system. Its values can be *no-flow*, *infinite* or *constant pressure*. The attributes **wellbore-storage** and **fractured** contain only "true" or "false" depending on the occurrence of such phenomena.

These five objects contain also graphic-related attributes. In ART, all graphics (windows, lines, rectangles, text,...) are also represented by objects and attributes. For example, a line object will have attributes such as **origin, endpoint, thickness**, etc. The attributes in Table 3-1 ending in **-curve** or **-icons** point to such objects.

3.2.2 Hypothetical Worlds

During the analysis, an expert is likely to follow several alternative interpretations, until he or she is eventually convinced that one is more likely than the others. The analysis may also end-up in a dead-lock, the expert concluding that more information is needed to reach a reliable conclusion.

WES models this kind of behavior by generating hypotheses about the actual model. These hypotheses are kept in separate "worlds". Each of these hypothetical worlds is the state of the data base resulting from one hypothesis, or one group of compatible hypotheses. The program maintains these worlds simultaneously, and rules of inference can be activated independently in each of them depending on the facts present in the particular state of the data base. Compatible hypotheses can be grouped together. For example, an hypothesis based on the early part of the data set can be grouped with an hypothesis on the intermediate part and an hypothesis on the late part, to form a complete hypothesis. The rules generating hypotheses are fairly weak, and as soon as there is a slight chance for a particular model to be true, the corresponding hypothesis is

generated. This is the way the system takes care of the inherent imprecision of the theory.

The basic data architecture is common to all the subsequent hypothetical worlds generated by the system. Typically, attributes in an instance of the **well-model** object will have different values in different hypothetical worlds. In one particular hypothetical world, each of those attributes can only have one value. For example, the **boundary** attribute cannot have values *no-flow* and *infinite* in the same world, but can take those two values in two different worlds.

This concept of hypothetical worlds is achieved by making use of the more general concept of *viewpoints* available in ART. The system uses one level of viewpoints, and each viewpoint represents one hypothetical world.

3.3 Rule Architecture and Runtime Behavior

Even though this is a rule based system, in which the flow of control should be by nature opportunistic, the system always proceeds sequentially through a number of steps. Within each of these steps, several rules are executable at the same time, depending on whether their left-hand-side (or "if" part) conditions are satisfied by facts in the current state of the system or not. The choice of the particular rule to be fired, or conflict resolution, is implemented in ART language.

The rules do not refer to a specific well, but rather can be executed "simultaneously" for more than one well so that parallel analysis of many wells is possible. In the long term, this feature will be useful for performing the analysis of multiple-well tests, i.e. tests in which water is pumped in or from a single well and several other wells are used for observation.

The different steps during the analysis of a single well are shown in Figure 3-2.

3.3.1 Data Extraction

The initial data is stored in a file. This data is composed of readings of pressure at different times. Time is represented in hours, minutes and seconds, and pressure in psi. Different units can also be used, but data will always be converted to seconds and psi.

Data extraction from a disc file

↓

Window and initial curve drawing

↓

Data filtering

↓

Pressure derivatives computation and validity checking

↓

Curve characteristics extraction on semilog, loglog and derivative plots

↓

Generation of hypotheses

↓

Selection of one or several models

Figure 3-2. Execution steps.

3.3.2 Graphics

Once the user has chosen to conduct an analysis on a specific well and as soon as the data set is read from disc, WES prompts for a window, which will be used for curve drawing, and for interactions between the user and the system. The user must then use the mouse to define the position and size of the window. One window is created for each well on which the user wants to perform an analysis. This window contains five mouse-clickable icons, which are: **initial, filtered, semilog, loglog** and **derivative**. The reasoning process on a specific well is started by clicking on the **filtered** icons in its window with the left button. After that, the five icons have the same function of displaying the corresponding curve inside the window. Also, by clicking with the mouse's right button, one can superimpose graphical representations of straight lines and humps (see Section 3.3.5) on the semilog, loglog and derivative plots (Figure 3-3).

3.3.3 Data Filtering

Since data sets can be very different in size and may sometimes contain a lot of data points, the system first filters the initial data set and selects a specified number of data points. This number is fixed by the user before execution. One data point on the filtered plot is the result of an averaging on the initial plot. From our experience so far, a "good" number of data points could be somewhere between fifty and a hundred. Too few data points are likely to hide some meaningful shapes, and too many slow down the execution and give noisy data to the reasoning phase of the analysis.

The abscissa for all the curves used in the analysis is the log of time. Therefore the system keeps a constant number of data points in each log cycle on the time scale. Since pressure data is generally recorded at fairly constant time intervals, the density of data points on a log scale increases dramatically with the time. The filtering process thus results often in a drastic reduction of the number of late data points, whereas the system keeps most the data points in the early part of the initial curve.

Besides discarding too numerous data points the filtering phase smoothes the data in order

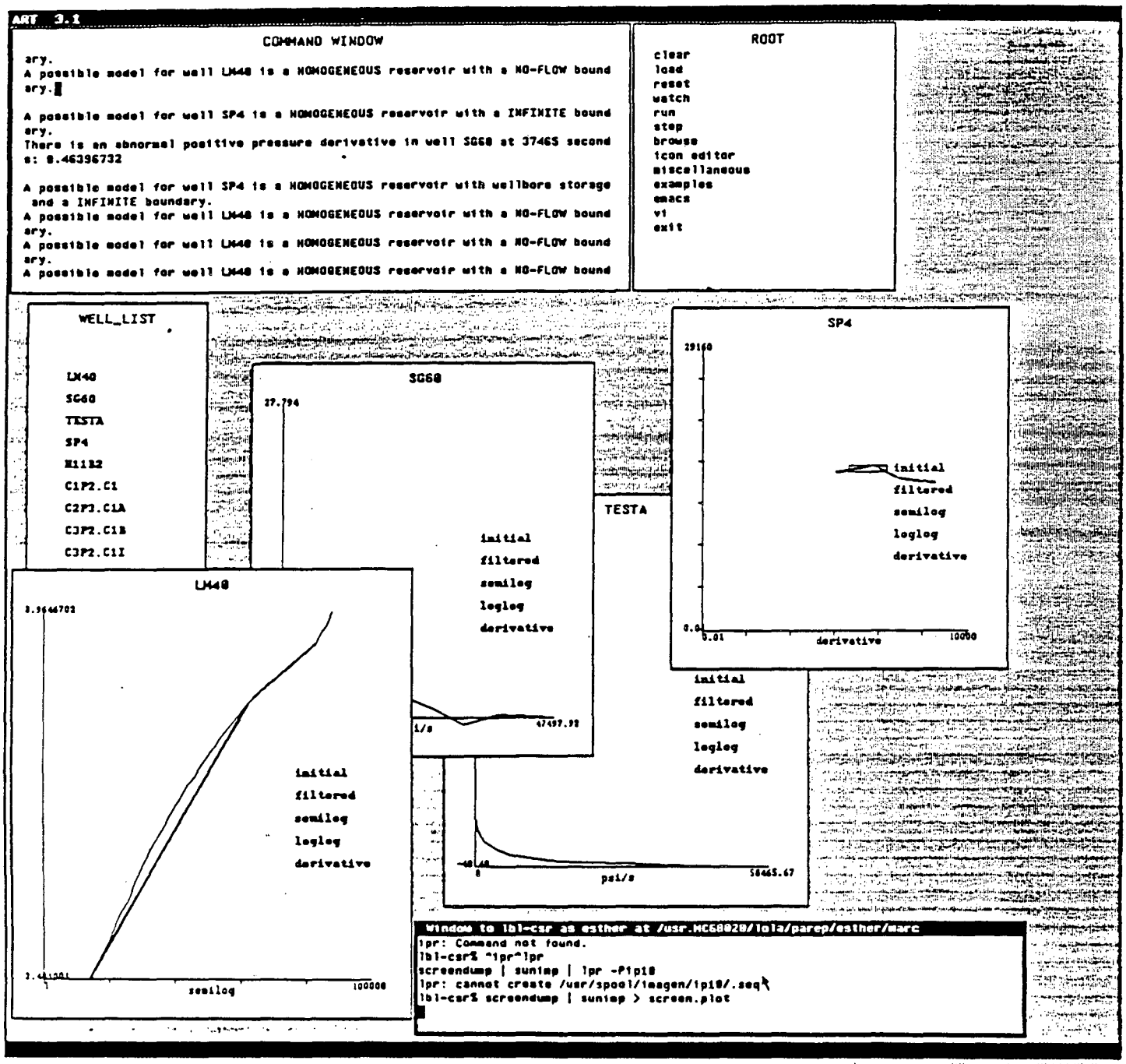


Figure 3-3. Example screen during a run of WES.

to avoid undesirable and meaningless noise.

3.3.4 Pressure Derivative

Most of the expertise used by WES to find conceptual models is applied to the pressure derivative curve. During this numerical step, WES computes the pressure derivative curve, which will thus serve as a basis for a large part of the interpretation. The program also computes the second order derivative and checks the validity of the first order derivative. Since the test under analysis is a drawdown, the pressure is always diminishing, thus the derivative must always be negative. In case the system finds a positive pressure derivative at some particular time, a warning message is displayed and the value is set to 0, or rather to -0.01, which is the biggest negative number that the system can admit. All negative numbers with an absolute value lower than -0.01 are also set back to -0.01.

The fact that there is a positive pressure derivative at some particular time is not normal and could have several different causes, such as deficient equipment, atmospheric pressure changes, non constant flow rate, etc. A logical step in such a case is to prompt the user for evidence of one of these various causes, and then if possible correct the curve using this new evidence. This analysis is not done by the prototype at this stage of development.

The algorithm used for computing derivatives is described in Section 3.4.1.

3.3.5 Aggregated Description of Curves

A human expert conducts his analysis by looking at global characteristics of curves. Well known models, like homogeneous and infinite systems, have different signatures on the derivative, log-log and semilog plots. This step in the execution of the program looks for signatures such as straight lines, up or down humps or values of slopes on the three different plots.

On the derivative plot, straight lines, humps, maxima and minima are searched for. The second order derivative gives the value of the slope at each point on the derivative curve. On semilog and log-log plots, only straight lines are computed in advance. When needed however, derivatives of those particular plots can be computed. Note that the slopes on the semilog plot

are already known since they correspond to the values on the derivative plot.

Straight lines are characterized by two points and a slope. Humps are defined by a word (*hill* or *valley*) telling whether the hump is positive or negative, a starting point, a maximum or a minimum, and an ending point. The algorithms used to extract the straight lines and humps from the curves are described in Section 3.4.

3.3.6 Generation of Possible Models

Partial conceptual models are generated from the descriptions of curves in terms of straight lines and humps. Rules try to represent the decision criteria used by the experts. An example of such a rule is:

> If there is a hill hump followed by a horizontal straight line on the derivative plot, then generate the hypothesis of a homogeneous and infinite system.

These rules can generate hypotheses based on small parts of the pressure versus time curve, for example wellbore storage from early time data, or the type of boundary from the late part of the curve. Those partial hypotheses can then be grouped together to form a complete model, such as wellbore storage in a homogeneous medium with a no flow boundary. The program can only group compatible partial models. For example, grouping an infinite system with a no flow boundary is not allowed. These incompatible hypotheses are kept separate in different hypothetical worlds.

3.3.7 Selection of One or More Models

In the present state of the program, all possible complete models are simply printed on the screen. A complete model is a model which gives at least the type of medium (homogeneous or double-porosity) and the kind of boundary (infinite, no-flow or constant-pressure). The early time characteristics, wellbore storage or fractured formation near the well, are optional. In the current implementation of WES, the same model may be printed more than once in some instances. It means that the system reached the same conclusion by two or more paths of reasoning. A model that appears more than once is thus more likely to be true than a model printed

only once.

In some cases, for some data sets, no models are proposed. It means either that the expertise present in the system is not sufficient to analyze this particular case or that this data set has some abnormal shape, due to some external event.

3.4 Numerical Algorithms

This section describes the algorithms used for computing derivatives and extracting straight lines and humps. These algorithms are implemented as LISP functions callable from ART.

3.4.1 Derivatives

Various algorithms have been tested for computing the pressure derivative. The derivative is taken with respect to the log of time. One of the main inconveniences of the pressure derivative approach is that it cannot be measured directly but rather must be computed from discrete data. A good algorithm must preserve all the meaningful response of the system while removing all or most of the noisy parts. The algorithm that is used currently is inspired from the one described in Bourdet et al. (1984) and corresponds to algorithm B in Clark et al. (1985). It may still be improved but it gives the best results among all the ones that have been tested. The LISP function that implements this algorithm is called *derivative-logtime.*

The algorithm computes the weighted mean of the slopes between the point under study and a point preceding it, and between the point under study and a point following it. The two points are not the points closest to the point of interest, but instead are defined by skipping several points to go from the point under study to the two points where we will take the slope. Since all the points are equally spaced on a log scale, this amounts in our case to using slopes at constant intervals from the point of interest. In the example on Figure 3-4 this number is 3. It is the number currently used by the system.

Using the notations in Figure 3-4, the slope p' is then given by:

$$p' = \frac{\frac{\Delta p1}{\Delta t1}\Delta t2 + \frac{\Delta p2}{\Delta t2}\Delta t1}{\Delta t1 + \Delta t2}$$

3.4.2 Straight Lines

The algorithm for extracting straight lines from a curve is based on the value of the slope p′ at each point. Thus, for the semilog, the pressure derivative is used, and for the pressure derivative, the second order derivative is used. The second order derivative p'' is computed in the same way as the first order. The exact formula is:

$$p'' = \frac{d \log \left[\frac{d\,p}{d \log t} \right]}{d \log t}$$

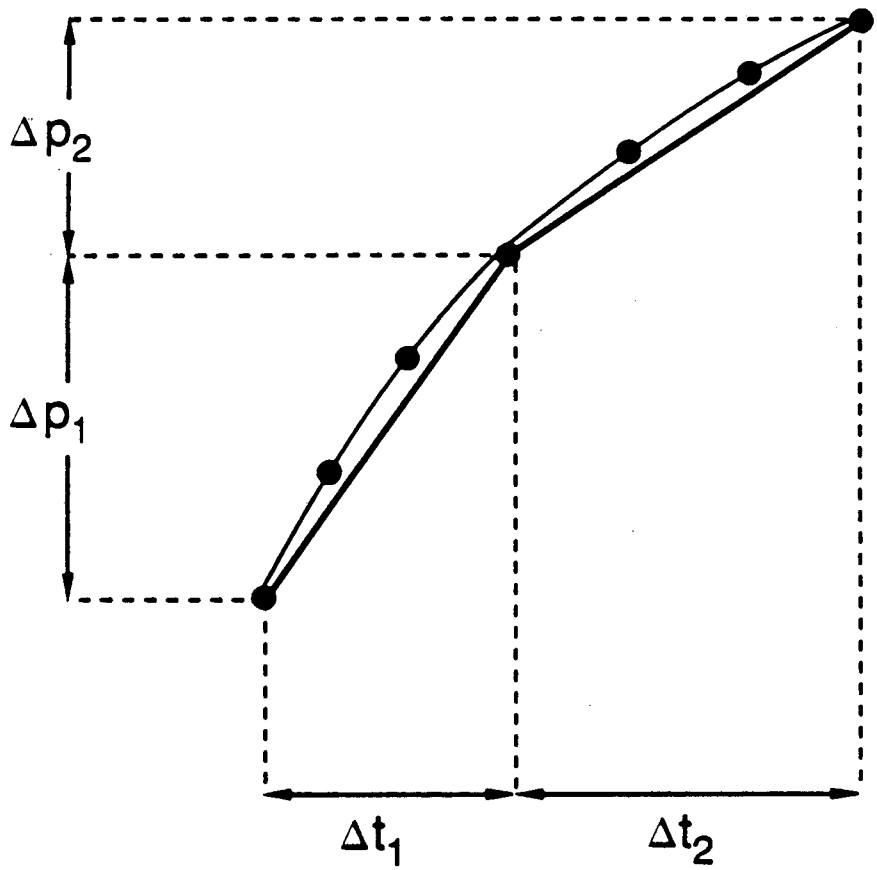
The algorithm is:

```
        i=0
1       j=i+1
2       if | p_j' - (Σ_{i≤k<j} p_k') / (j-i) | ≤ error
        then  line = [ t_i, t_j ]
              j=j+1
              if end then goto 3 else goto 2
        else  if length(line) > 1/4 log cycle then return line
          i=j
              goto 1
3       if length(line) > 1/4 log cycle then return line
```

The formula in step 2 of the algorithm above states that a data point will belong to the straight line under consideration if and only if its derivative is equal, within a given tolerance, to the average of the derivatives at all the points already included in the straight line.

3.4.3 Humps

Humps are identified on the derivative plot only. The algorithm used here was chosen because it is not overly sensitive to small peaks, and it disregards shapes shorter than a quarter of a log cycle. As in the straight line algorithm, the minimum length of a quarter of a log cycle is a



XBL 882-10085

Figure 3-4. Notations for computing derivatives.

parameter defined in ART and can be reset before each execution.

The following algorithm recognizes only negative humps (*hill* humps). The algorithm for positive humps is symmetrical to this one and the two are actually implemented as one LISP function, which returns whether the hump identified is a *hill* or a *valley* hump. Note that on the derivative plot, p'' represents the slope. The notation [ ] represents an interval on the curve defined by its starting and ending times. Lengths are computed as portions of a log-cycle.

```
        i=0
1       if p_i' ≥ 0 then go to 2 else i=i+1; go to 1      start of hill hump must have positive slope
2       j=i                                               start of a hump
        noise = [ t_j, t_j ]                              initialize "noisy" interval to zero length
3       if t_j+1 exists
        and length(noise) ≤ 1/4 log cycle                 some "noise" admitted within the hump
        then if p_i'' ≥ p_j'' ≥ p_j+1''                   if "hill" curvature
            then before–noise = t_j                       reinitialize beginning of noise for latter use
                noise = [ t_j, t_j ]                      reinitialize "noisy" interval to zero length
                hump = [ t_i, t_j ]                       extend hump to [initial point, current point] interval
                if p_j''*p_j+1'' < 0 then top = t_j       if slope changes sign, we are at the maximum
            else noise = [ before–noise, t_j ]            extend noisy interval to [before-noise, current]
            endif
            j = j+1                                       increment current time
            go to 3
        else if length (hump) ≥ 3/4 log cycle             if hump long enough
            and length ([t_i,top]) ≥ 1/4 log cycle        if part before the extremum long enough
            and length ([top,t_j]) ≥ 1/4 log cycle        if part after the extremum long enough
            then return hump: t_i, t_j, top               specify hump by [start, end, maximum]
            if t_j+1 does not exist then stop             end of the curve
        endif
        i = j                                             reinitialize search for start of hump
        go to 1
```

In the current implementation, the length of a hump must be at least 3 quarters of a log cycle, and the length on each side of the extremum must be at least one quarter of a log cycle.

3.5 Implementation: the ART Programming Tool

WES is written in ART, the Automated Reasoning Tool from Inference Corp. The main advantages of ART for building this kind of expert system are:

(1) its object oriented representation of data, which allows to conduct many analyses at the same time without interference between the analysis;

(2) its sophisticated pattern recognition algorithm, which allows complex conditions and variables in the left hand side of rules;

(3) the viewpoint mechanism for reasoning with hypotheses;

(4) its coupling with LISP, which was necessary for the numerical part of the problem.

Although the viewpoint structure is satisfactory for representing hypotheses and reasoning with them, it is not certain that it will be sufficient in the future to tackle all the subtleties of well testing. The problem with this domain is the inherent imprecision of the theory. Many times during the development of the system, it was necessary to define numerical thresholds. The crisp values given to those thresholds give the system a kind of rigidity which does not exist in the human expert's mind.

The version of ART that was used is version 3.1 on a Sun 280. The listing of the program is given in the appendix. It has four different parts, which correspond to four different files. The first one defines the object hierarchy and the graphics interface for curves and icons. The second part computes derivatives and looks for straight lines and humps. The third part proposes some models according to some curve characteristics. The last one is written in LISP and contains the LISP functions used throughout the program.

3.6 Possible Extensions

3.6.1 Extending the Analysis of the Present Problem

The present system is in the early stages of development. A lot of work needs to be done in the third part of the program, which contains the most expertise of the domain. So far the system has limited notions of simple models, like homogeneous or simple double porosity transition and some boundary configurations. More models need to be inserted as well as a more complete description of the existing ones.

In the same respect, little expertise is used currently during the extraction of curve signatures. This part of the processing is presently performed as a purely numerical procedure. In fact, it is not separate from the analysis part. For example, the expert very often knows that a given shape is impossible, either in general or in regard with what he already knows about the system. The fact that the shape identification procedures are controlled by rules will make is relatively easy to reproduce this kind of expertise. Also in the current version, the generation of hypotheses is restricted to models. It can very well be extended to permit the characteristics of the curves to be also considered as hypotheses.

3.6.2 Extending to a Larger Problem

In the present state of the system, only single wells with a constant flow rate and a drawdown phase are analyzed. The next step will be to allow a buildup phase and more generally a multiple rate pumping phase. The system should be able to reason about these different phases, compare results between them, decide which phase is more informative than the others, and so on. Another step in the development of the system will be to allow more than one observation well for a single pumping well.

The main problem in the analysis is that the data is often very noisy. In order to get rid of this noise, more information needs to be known, or at least to be accessible by the system. For example, an expert that sees a very strange shape will look at other sources of informations like atmospheric pressure changes, geologic data, equipment used, and so on, and will try to explain

those abnormal shapes. More sophisticated rationales for identifying these types of influences are needed before the reasoning procedure can readily be implemented for such cases.

4. REFERENCES

Bachant, J., J. McDermott (1984). R1 revisited: Four Years in the Trenches. *AI Magazine, 5,* (3) 21-32.

Bobrow, D. G., S. Mittal, M. J. Stefik (1986). Expert Systems: Perils and Promise. *Communications of the ACM, 29,* (9) 880-894.

Bourdet, D., J. A. Ayoub, Y. M. Pirard (1984). Use of Pressure Derivative in Well Test Interpretation. Paper SPE 12777 presented at the 1984 SPE California Regional Meeting, Long Beach, April 11-13, 1984.

Clark, D. G., T. D. Van Golf-Racht (1985). Pressure-Derivative Approach to Transient Test Analysis: A High-Permeability North Sea Reservoir Example, *Journal of Petroleum Technology,* November, 2023-2039.

Clayton, B. D. (1984). User Manuals for ART, the Automated Reasoning Tool. Inference Corp., Los Angeles.

Denning, P. J. (1986). Towards a Science of Expert Systems. *IEEE Expert,* (1) 80-83.

McDermott, J. (1980). R1: An Expert in the Computer Systems Domain. *AAAI,* (1) 269-271.

McDermott, J. (1981). R1: The formative years. *AI Magazine, 2,* (2) 21-29.

Harmon, P., D. King (1985). *Expert systems - Artificial Intelligence in Business.* John Wiley and Sons, Inc., New York, 282 pp.

Hayes-Roth, F., D. A. Waterman, D. B. Lenat, editors (1983). *Building Expert Systems,* Addison Wesley, Reading, Massachusetts, 444 pp.

Weiss, S. M., C. A. Kulikowski (1984). *A Practical Guide to Designing Expert Systems,* Rowman and Allanheld, Totowa, N.J.

APPENDIX

- Listing of the Program -

```
;;; -*- mode: ART; Package: ART-USER; Base:10. -*-
;;; file: well/init.art

;;; This file contains the initial facts and schemata and all the rules
;;; used for windows and icons definition and drawing.

;;; The program will use a one level viewpoint structure. This level,
;;; called "hypothetical" will contain all hypothesis regarding the
;;; nature of the medium.

(def-viewpoint-levels hypothetical)

;;; Initial facts and relations.

(defrelation data-set (?well)
        "well with available data")
(defrelation interval-number (?number)
        "number of data points used for the analysis")
(defrelation abs-d-error (?error)
        "absolute imprecision allowed on the derivatives")
(defrelation rel-d-error (?rel)
        "relative imprecision allowed on the derivatives")
(defrelation analyse (?well)
        "well on which the analysis will be conducted" implicit)
(defrelation p-window (?l)
        "width used to compute the derivative")
(defrelation p-d-window (?l)
        "width used to compute the derivative of the p-d")
(defrelation derivative-function (?function)
        "function used to compute derivatives")
(defrelation significant-length (?sig-length)
        "length in fraction of a log cycle")

(deffacts data "Pumping Wells"
        (data-set C3P2.C1I)
        (data-set C3P2.C1B)
        (data-set C2P3.C1A)
        (data-set C1P2.C1)
        (data-set H11B2)
        (data-set SP4)
        (data-set TESTA)
        (data-set SG60)
        (data-set LM40)
        )

(deffacts parameters "used for curve analysis"
        (interval-number 60)
        (abs-d-error 0.15)
        (rel-d-error 0.20)
        (p-window 3)
        (p-d-window 3)
        (derivative-function derivative-logtime)
        (significant-length 0.25)
        )

(defschema time
        (instance-of slot)
        (slot-derivation implicit))

(defschema pressure
        (instance-of slot)
        (slot-derivation implicit))

(defschema filtered-curve
        (instance-of slot)
        (slot-derivation implicit))
```

```
(defschema semilog-curve
        (instance-of slot)
        (slot-derivation implicit))

(defschema loglog-curve
        (instance-of slot)
        (slot-derivation implicit))

(defschema derivative
        (instance-of relation)
        (inverse derivative-of))

(defschema semilog
        (instance-of relation)
        (inverse semilog-of))

(defschema loglog
        (instance-of relation)
        (inverse loglog-of))

(defschema model
        (instance-of relation)
        (inverse model-of))

(defschema straight-line
        (instance-of slot)
        (slot-how-many multiple-values))

(defschema hump
        (instance-of slot)
        (slot-how-many multiple-values))

(defschema examined
        (instance-of slot)
        (slot-how-many multiple-values))

(defschema derivative-icons
        (instance-of slot)
        (slot-how-many multiple-values))

(defschema semilog-icons
        (instance-of slot)
        (slot-how-many multiple-values))

(defschema loglog-icons
        (instance-of slot)
        (slot-how-many multiple-values))

(defschema well
        (initial-time)
        (initial-pressure)
        (time)
        (pressure)
        (derivative)
        (semilog)
        (loglog)
        (model)
        (i-last-time)
        (i-last-pressure)
        (last-time)
        (first-pressure)
        (last-pressure)
        (x-scale)
        (y-scale)
        (initial-curve)
```

```
        (filtered-curve)
        (common-icons)
        (is-a instantiated-window-icon)
        (display-parameters)
        (window)
        )

(defschema well-semilog
        (semilog-of)
        (straight-line)
        (semilog-curve)
        (semilog-icons)
        )

(defschema well-loglog
        (loglog-of)
        (log-derivative)
        (straight-line)
        (loglog-curve)
        (loglog-icons)
        )

(defschema well-derivative
        (derivative-of)
        (time)
        (p-derivative)
        (examined)
        (p-d-derivative)
        (first-p-derivative)
        (max-p-derivative)
        (min-p-derivative)
        (max-p-d-derivative)
        (min-p-d-derivative)
        (straight-line)
        (hump)
        (derivative-curve)
        (derivative-icons)
        )

(defschema well-model
        (model-of)
        (reservoir)
        (early)
        (intermediate)
        (late)
        (wellbore-storage)
        (fractured)
        (boundary)
        )

(defrule initial-data
        (data-set ?well)
        (analyse ?well)
=>
        (write-string "Reading data for well ")
        (write-string (string ?well))
        (write-line "...")
        (bind ?data (read-from-file ?well))
        (assert (schema ?well
                        (instance-of well)
                        (initial-time =(first ?data))
                        (initial-pressure =(second ?data))
                        (i-last-time =(third ?data))
                        (first-pressure =(fourth ?data))
                        (i-last-pressure =(fifth ?data))
                        (derivative =(concat ?well 'derivative))
```

```
                        (semilog =(concat ?well 'semilog))
                        (loglog =(concat ?well 'loglog))))
        (write-line "Open a window, please. (Curve display).")
        (create-window ?well 'graphics)
        (assert (x-scale ?well =(- (window-inside-width ?well) 50)))
        (assert (y-scale ?well =(- (window-inside-height ?well) 50)))
        )

(defrule ask-for-analysis
        (goal (analyse ?well&~??))
=>
        (if (y-or-n-p (concatenate 'string
                        "Would you like to proceed with the analysis of well "
                        (write-to-string ?well) " ? "))
         then (write-line "")
              (assert (analyse ?well)))
        )

(defrule compute-and-draw-initial-curve
        (data-set ?well)
        (initial-time ?well ?time)
        (initial-pressure ?well ?pressure)
        (first-pressure ?well ?first-p)
        (i-last-pressure ?well ?last-p)
        (i-last-time ?well ?last-t)
        (x-scale ?well ?x)
        (y-scale ?well ?y)
=>
        (bind ?yscale (concat 'yscale ?well))
        (bind ?xscale (concat 'xscale ?well))
        (bind ?curve-i (concat 'curve-i ?well))
        (bind ?text-i0 (concat 'text-i0 ?well))
        (bind ?text-i1 (concat 'text-i1 ?well))
        (bind ?text-i2 (concat 'text-i2 ?well))
        (bind ?text-i3 (concat 'text-i3 ?well))
        (bind ?text-i4 (concat 'text-i4 ?well))
        (bind ?i-but (concat 'i-but ?well))
        (bind ?f-but (concat 'f-but ?well))
        (bind ?s-but (concat 's-but ?well))
        (bind ?l-but (concat 'l-but ?well))
        (bind ?d-but (concat 'd-but ?well))
        (assert (initial-curve ?well (?curve-i ?text-i0 ?text-i1 ?text-i2
                                      ?text-i3 ?text-i4)))
        (assert (common-icons ?well (?yscale ?xscale ?i-but ?f-but ?s-but
                                     ?l-but ?d-but)))
        (bind ?diff-p (?last-p - ?first-p))
        (bind ?xp (compute-coordinates ?x 0 ?last-t (list$ ?time)))
        (bind ?yp (compute-coordinates ?y ?first-p ?diff-p (list$ ?pressure)))
        (assert (schema ?curve-i
                        (instance-of open-polygon)
                        (alu ior)
                        (translate (20 20))
                        (x-points =(cond ((> (list-length ?xp) 256)
                                          (nbutlast ?xp
                                                    (- (list-length ?xp) 256)))
                                         (t ?xp)))
                        (y-points =(cond ((> (list-length ?yp) 256)
                                          (nbutlast ?yp
                                                    (- (list-length ?yp) 256)))
                                         (t ?yp)))
                        (thickness 1)))
        (assert (schema ?yscale
                        (instance-of line)
                        (alu ior)
                        (translate (20 20))
                        (endpoint (0 ?y))
```

```
                (thickness 1)))
(assert (schema ?text-i0
                (instance-of text)
                (alu ior)
                (translate (0 20))
                (text-string ?first-p)
                (endpoint (10 0))
                (font font-8)))
(assert (schema ?text-i1
                (instance-of text)
                (alu ior)
                (translate (0 =(+ ?y 20)))
                (text-string ?last-p)
                (endpoint (10 0))
                (font font-8)))
(assert (schema ?xscale
                (instance-of line)
                (alu ior)
                (translate (20 =(+ ?y 20)))
                (endpoint (?x 0))
                (thickness 1)))
(assert (schema ?text-i2
                (instance-of text)
                (alu ior)
                (translate (20 =(+ ?y 30)))
                (text-string "0")
                (endpoint (10 0))
                (font font-8)))
(assert (schema ?text-i3
                (instance-of text)
                (alu ior)
                (translate (?x =(+ ?y 30)))
                (text-string ?last-t)
                (endpoint (10 0))
                (font font-8)))
(assert (schema ?text-i4
                (instance-of text)
                (alu ior)
                (translate (=(/ ?x 2) =(+ ?y 35)))
                (text-string "psi/s")
                (endpoint (10 0))
                (font font-9)))
(assert (schema ?i-but
                (instance-of text)
                (input ((initial) "Display initial curve."))
                (alu ior)
                (contained-in-icon ?well)
                (translate (=(- ?x 60) =(/ ?y 2)))
                (text-string "initial")
                (endpoint (10 0))
                (font font-6)))
(assert (schema ?f-but
                (instance-of text)
                (input ((filtered) "Filter data and start reasoning process on thi
                (alu ior)
                (contained-in-icon ?well)
                (translate (=(- ?x 60) =(+ (/ ?y 2) 25)))
                (text-string "filtered")
                (endpoint (10 0))
                (font font-6)))
(assert (schema ?s-but
                (instance-of text)
                (input ((semilog) "Display semilog curve. L: Alone. R: With straig
                (alu ior)
                (contained-in-icon ?well)
                (translate (=(- ?x 60) =(+ (/ ?y 2) 50)))
```

```
                        (text-string "semilog")
                        (endpoint (10 0))
                        (font font-6)))
        (assert (schema ?l-but
                        (instance-of text)
                        (input ((loglog) "Display loglog curve. L: Alone. R: With straight
                        (alu ior)
                        (contained-in-icon ?well)
                        (translate (=(- ?x 60) =(+ (/ ?y 2) 75)))
                        (text-string "loglog")
                        (endpoint (10 0))
                        (font font-6)))
        (assert (schema ?d-but
                        (instance-of text)
                        (input ((derivative)
                                "Display derivative curve. L: Alone. R: With straight line
                        (alu ior)
                        (contained-in-icon ?well)
                        (translate (=(- ?x 60) =(+ (/ ?y 2) 100)))
                        (text-string "derivative")
                        (endpoint (10 0))
                        (font font-6)))
        (assert (schema ?well
                (contains-icons (?yscale ?xscale ?curve-i ?text-i0
                                 ?text-i1 ?text-i2 ?text-i3 ?text-i4
                                 ?i-but ?f-but ?s-but ?l-but ?d-but))))
        (show-icon ?well ?well)
)

(defrule compute-filtered-curve
        (goal (filtered-curve ?well&~?? ??))
        (not (filtered-curve ?well ?))
        (time ?well ?time)
        (pressure ?well ?pressure)
        (first-pressure ?well ?first-p)
        (last-pressure ?well ?last-p)
        (last-time ?well ?last-t)
        (x-scale ?well ?x)
        (y-scale ?well ?y)
=>
        (bind ?curve-f (concat 'curve-f ?well))
        (bind ?text-f0 (concat 'text-f0 ?well))
        (bind ?text-f1 (concat 'text-f1 ?well))
        (bind ?text-f2 (concat 'text-f2 ?well))
        (bind ?text-f3 (concat 'text-f3 ?well))
        (bind ?text-f4 (concat 'text-f4 ?well))
        (assert (filtered-curve ?well (?curve-f ?text-f0 ?text-f1 ?text-f2
                                 ?text-f3 ?text-f4 )))
        (bind ?diff-p (?last-p - ?first-p))
        (assert (schema ?curve-f
                        (instance-of open-polygon)
                        (alu ior)
                        (translate (20 20))
                        (x-points =(compute-coordinates ?x 0 ?last-t
                                                        (list$ ?time)))
                        (y-points =(compute-coordinates ?y ?first-p ?diff-p
                                                        (list$ ?pressure)))
                        (thickness 1)))
        (assert (schema ?text-f0
                        (instance-of text)
                        (alu ior)
                        (translate (0 20))
                        (text-string ?first-p)
                        (endpoint (10 0))
                        (font font-8)))
        (assert (schema ?text-f1
```

```
                        (instance-of text)
                        (alu ior)
                        (translate (0 =(+ ?y 20)))
                        (text-string ?last-p)
                        (endpoint (10 0))
                        (font font-8)))
        (assert (schema ?text-f2
                        (instance-of text)
                        (alu ior)
                        (translate (20 =(+ ?y 30)))
                        (text-string "0")
                        (endpoint (10 0))
                        (font font-8)))
        (assert (schema ?text-f3
                        (instance-of text)
                        (alu ior)
                        (translate (?x =(+ ?y 30)))
                        (text-string ?last-t)
                        (endpoint (10 0))
                        (font font-8)))
        (assert (schema ?text-f4
                        (instance-of text)
                        (alu ior)
                        (translate (=(/ ?x 2) =(+ ?y 35)))
                        (text-string "psi/s")
                        (endpoint (10 0))
                        (font font-9)))
        )


(defrule compute-semilog-curve
        (goal (semilog-curve ?well-s&~?? ??))
        (not (semilog-curve ?well-s ?))
        (semilog-of ?well-s ?well)
        (time ?well ?time)
        (pressure ?well ?pressure)
        (first-pressure ?well ?first-p)
        (last-pressure ?well ?last-p)
        (last-time ?well ?last-t)
        (x-scale ?well ?x)
        (y-scale ?well ?y)
=>
        (bind ?curve-s (concat 'curve-s ?well))
        (bind ?text-s0 (concat 'text-s0 ?well))
        (bind ?text-s1 (concat 'text-s1 ?well))
        (bind ?text-s2 (concat 'text-s2 ?well))
        (bind ?text-s3 (concat 'text-s3 ?well))
        (bind ?text-s4 (concat 'text-s4 ?well))
        (bind ?tick-sx (concat 'tick-sx ?well))
        (assert (semilog-curve ?well-s (?curve-s ?text-s0 ?text-s1 ?text-s2
                                        ?text-s3 ?text-s4 ?tick-sx)))
        (bind ?diff-p (?last-p - (nth$ ?pressure 2)))
        (bind ?t0 1)
        (bind ?diff-t (ceiling (log ?last-t 10)))
        (bind ?p0 (- ?first-p (nth$ ?pressure 2)))
        (assert (schema ?curve-s
                        (instance-of open-polygon)
                        (alu ior)
                        (translate (20 =(+ ?y 20)))
                        (x-points =(compute-coordinates ?x (log ?t0 10)
                                        ?diff-t
                                        (log-of-list (cdr (list$ ?time)))))
                        (y-points =(compute-coordinates ?y ?p0 ?diff-p
                                                (delta (list$ ?pressure))))
                        (thickness 1)))
        (assert (schema ?text-s0
```

```
                        (instance-of text)
                        (alu ior)
                        (translate (0 20))
                        (text-string =(- ?first-p ?last-p))
                        (endpoint (10 0))
                        (font font-8)))
        (assert (schema ?text-s1
                        (instance-of text)
                        (alu ior)
                        (translate (0 =(+ ?y 20)))
                        (text-string ?p0)
                        (endpoint (10 0))
                        (font font-8)))
        (assert (schema ?text-s2
                        (instance-of text)
                        (alu ior)
                        (translate (20 =(+ ?y 30)))
                        (text-string ?t0)
                        (endpoint (10 0))
                        (font font-8)))
        (assert (schema ?text-s3
                        (instance-of text)
                        (alu ior)
                        (translate (?x =(+ ?y 30)))
                        (text-string =(expt 10 ?diff-t))
                        (endpoint (10 0))
                        (font font-8)))
        (assert (schema ?text-s4
                        (instance-of text)
                        (alu ior)
                        (translate (=(/ ?x 2) =(+ ?y 35)))
                        (text-string "semilog")
                        (endpoint (10 0))
                        (font font-9)))
        (assert (schema ?tick-sx
                        (instance-of tick-marks)
                        (alu ior)
                        (translate (20 =(+ ?y 20)))
                        (endpoint (?x 5))
                        (direction vertical)
                        (interval =(/ ?x ?diff-t))
                        (thickness 1)))
)

(defrule compute-loglog-curve
        (goal (loglog-curve ?well-l&~?? ??))
        (not (loglog-curve ?well-l ?))
        (loglog-of ?well-l ?well)
        (time ?well ?time)
        (pressure ?well ?pressure)
        (last-time ?well ?last-t)
        (x-scale ?well ?x)
        (y-scale ?well ?y)
=>
        (bind ?curve-l (concat 'curve-l ?well))
        (bind ?text-l0 (concat 'text-l0 ?well))
        (bind ?text-l1 (concat 'text-l1 ?well))
        (bind ?text-l2 (concat 'text-l2 ?well))
        (bind ?text-l3 (concat 'text-l3 ?well))
        (bind ?text-l4 (concat 'text-l4 ?well))
        (bind ?tick-lx (concat 'tick-lx ?well))
        (bind ?tick-ly (concat 'tick-ly ?well))
        (assert (loglog-curve ?well-l (?curve-l ?text-l0 ?text-l1 ?text-l2
                                       ?text-l3 ?text-l4 ?tick-lx ?tick-ly)))
        (bind ?t0 0.01)
        (bind ?diff-t (- (ceiling (log ?last-t 10)) (log ?t0 10)))
```

```
(bind ?p0 0.01)
(assert (schema ?curve-1
                (instance-of open-polygon)
                (alu ior)
                (translate (20 =(+ ?y 20)))
                (x-points =(compute-coordinates ?x
                                (log ?t0 10)
                                ?diff-t
                                (log-of-list (cdr (list$ ?time)))))
                (y-points =(compute-coordinates ?x
                                (log ?p0 10)
                                (- ?diff-t)
                                (log-delta (list$ ?pressure))))
                (thickness 1)))
(assert (schema ?text-10
                (instance-of text)
                (alu ior)
                (translate (0 20))
                (text-string
                 =(round
                   (expt 10 (/ (* ?y (ceiling (log ?last-t 10))) ?x))))
                (endpoint (10 0))
                (font font-8)))
(assert (schema ?text-11
                (instance-of text)
                (alu ior)
                (translate (0 =(+ ?y 20)))
                (text-string ?p0)
                (endpoint (10 0))
                (font font-8)))
(assert (schema ?text-12
                (instance-of text)
                (alu ior)
                (translate (20 =(+ ?y 30)))
                (text-string ?t0)
                (endpoint (10 0))
                (font font-8)))
(assert (schema ?text-13
                (instance-of text)
                (alu ior)
                (translate (?x =(+ ?y 30)))
                (text-string =(expt 10 (ceiling (log ?last-t 10))))
                (endpoint (10 0))
                (font font-8)))
(assert (schema ?text-14
                (instance-of text)
                (alu ior)
                (translate (=(/ ?x 2) =(+ ?y 35)))
                (text-string "loglog")
                (endpoint (10 0))
                (font font-9)))
(assert (schema ?tick-1x
                (instance-of tick-marks)
                (alu ior)
                (translate (20 =(+ ?y 20)))
                (endpoint (?x 5))
                (direction vertical)
                (interval =(/ ?x ?diff-t))
                (thickness 1)))
(assert (schema ?tick-1y
                (instance-of tick-marks)
                (alu ior)
                (translate (15 =(+ 20 (mod ?y (/ ?x ?diff-t)))))
                (endpoint (5 ?y))
                (direction horizontal)
                (interval =(/ ?x ?diff-t))
```

```
                        (thickness 1)))
)

(defrule compute-derivative-curve
        (examined ?well-d yes)
        (p-derivative ?well-d ?p-deriv)
        (time ?well-d ?time)
        (derivative-of ?well-d ?well)
        (last-time ?well ?last-t)
        (x-scale ?well ?x)
        (y-scale ?well ?y)
=>
        (bind ?curve-d (concat 'curve-d ?well))
        (bind ?text-d0 (concat 'text-d0 ?well))
        (bind ?text-d1 (concat 'text-d1 ?well))
        (bind ?text-d2 (concat 'text-d2 ?well))
        (bind ?text-d3 (concat 'text-d3 ?well))
        (bind ?text-d4 (concat 'text-d4 ?well))
        (bind ?tick-dx (concat 'tick-dx ?well))
        (bind ?tick-dy (concat 'tick-dy ?well))
        (assert (derivative-curve ?well-d (?curve-d ?text-d0 ?text-d1 ?text-d2
                                           ?text-d3 ?text-d4 ?tick-dx ?tick-dy)))
        (bind ?t0 0.01)
        (bind ?diff-t (- (ceiling (log ?last-t 10)) (log ?t0 10)))
        (bind ?p0 0.01)
        (assert (schema ?curve-d
                        (instance-of open-polygon)
                        (alu ior)
                        (translate (20 =(+ ?y 20)))
                        (x-points =(compute-coordinates ?x
                                                (log ?t0 10)
                                                ?diff-t
                                                (log-of-list (list$ ?time))))
                        (y-points =(compute-coordinates ?x
                                        (log ?p0 10)
                                        (- ?diff-t)
                                        (log-of-list (list$ ?p-deriv))))
                        (thickness 1)))
        (assert (schema ?text-d0
                        (instance-of text)
                        (alu ior)
                        (translate (0 20))
                        (text-string
                         =(round
                           (expt 10 (/ (* ?y (ceiling (log ?last-t 10))) ?x))))
                        (endpoint (10 0))
                        (font font-8)))
        (assert (schema ?text-d1
                        (instance-of text)
                        (alu ior)
                        (translate (0 =(+ ?y 20)))
                        (text-string ?p0)
                        (endpoint (10 0))
                        (font font-8)))
        (assert (schema ?text-d2
                        (instance-of text)
                        (alu ior)
                        (translate (20 =(+ ?y 30)))
                        (text-string ?t0)
                        (endpoint (10 0))
                        (font font-8)))
        (assert (schema ?text-d3
                        (instance-of text)
                        (alu ior)
                        (translate (?x =(+ ?y 30)))
                        (text-string =(expt 10 (ceiling (log ?last-t 10))))
```

```
                        (endpoint (10 0))
                        (font font-8)))
        (assert (schema ?text-d4
                        (instance-of text)
                        (alu ior)
                        (translate (=(/ ?x 2) =(+ ?y 35)))
                        (text-string "derivative")
                        (endpoint (10 0))
                        (font font-9)))
        (assert (schema ?tick-dx
                        (instance-of tick-marks)
                        (alu ior)
                        (translate (20 =(+ ?y 20)))
                        (endpoint (?x 5))
                        (direction vertical)
                        (interval =(/ ?x ?diff-t))
                        (thickness 1)))
        (assert (schema ?tick-dy
                        (instance-of tick-marks)
                        (alu ior)
                        (translate (15 =(+ 20 (mod ?y (/ ?x ?diff-t)))))
                        (endpoint (5 ?y))
                        (direction horizontal)
                        (interval =(/ ?x ?diff-t))
                        (thickness 1)))
)

(defrule draw-initial-curve
  ?d <- (utterance ? ?well (mouse-click left 1 ? ? (initial)))
        (initial-curve ?well ?i-icons)
        (common-icons ?well ?c-icons)
=>
        (retract ?d)
        (bind ?icons (append (list$ ?c-icons) (list$ ?i-icons)))
        (modify (schema ?well
                        (contains-icons ?icons)))
        (show-icon ?well ?well)
)

(defrule draw-filtered-curve
  ?d <- (utterance ? ?well (mouse-click left 1 ? ? (filtered)))
        (filtered-curve ?well ?f-icons)
        (common-icons ?well ?c-icons)
=>
        (retract ?d)
        (bind ?icons (append (list$ ?c-icons) (list$ ?f-icons)))
        (modify (schema ?well
                        (contains-icons ?icons)))
        (show-icon ?well ?well)
)


(defrule draw-semilog-curve
  ?d <- (utterance ? ?well (mouse-click left 1 ? ? (semilog)))
        (semilog ?well ?well-s)
        (semilog-curve ?well-s ?s-icons)
        (common-icons ?well ?c-icons)
=>
        (retract ?d)
        (bind ?icons (append (list$ ?c-icons) (list$ ?s-icons)))
        (modify (schema ?well
                        (contains-icons ?icons)))
        (show-icon ?well ?well)
)

(defrule draw-loglog-curve
```

```
  ?d <- (utterance ? ?well (mouse-click left 1 ? ? (loglog)))
        (loglog ?well ?well-l)
        (loglog-curve ?well-l ?l-icons)
        (common-icons ?well ?c-icons)
=>
        (retract ?d)
        (bind ?icons (append (list$ ?c-icons) (list$ ?l-icons)))
        (modify (schema ?well
                        (contains-icons ?icons)))
        (show-icon ?well ?well)
)

(defrule draw-derivative-curve
  ?d <- (utterance ? ?well (mouse-click left 1 ? ? (derivative)))
        (derivative ?well ?well-d)
        (derivative-curve ?well-d ?d-icons)
        (common-icons ?well ?c-icons)
=>
        (retract ?d)
        (bind ?icons (append (list$ ?c-icons) (list$ ?d-icons)))
        (modify (schema ?well
                        (contains-icons ?icons)))
        (show-icon ?well ?well)
)

(defrule draw-derivative-icons
  ?d <- (utterance ? ?well (mouse-click right 1 ? ? (derivative)))
        (derivative ?well ?well-d)
        (exists (derivative-icons ?well-d ?))
        (derivative-curve ?well-d ?d-curve)
        (common-icons ?well ?c-icons)
=>
        (retract ?d)
        (bind ?d-icons (get-schema-value ?well-d 'derivative-icons))
        (bind ?icons (append (list$ ?c-icons)
                             (list$ ?d-curve)
                             (list$ ?d-icons)))
        (modify (schema ?well
                        (contains-icons ?icons)))
        (show-icon ?well ?well)
)

(defrule draw-semilog-icons
  ?d <- (utterance ? ?well (mouse-click right 1 ? ? (semilog)))
        (semilog ?well ?well-s)
        (exists (semilog-icons ?well-s ?))
        (semilog-curve ?well-s ?s-curve)
        (common-icons ?well ?c-icons)
=>
        (retract ?d)
        (bind ?s-icons (get-schema-value ?well-s 'semilog-icons))
        (bind ?icons (append (list$ ?c-icons)
                             (list$ ?s-curve)
                             (list$ ?s-icons)))
        (modify (schema ?well
                        (contains-icons ?icons)))
        (show-icon ?well ?well)
)

(defrule draw-loglog-icons
  ?d <- (utterance ? ?well (mouse-click right 1 ? ? (loglog)))
        (loglog ?well ?well-l)
        (exists (loglog-icons ?well-d ?))
        (loglog-curve ?well-l ?l-curve)
        (common-icons ?well ?c-icons)
=>
```

```
        (retract ?d)
        (bind ?l-icons (get-schema-value ?well-l 'loglog-icons))
        (bind ?icons (append (list$ ?c-icons)
                             (list$ ?l-curve)
                             (list$ ?l-icons)))
        (modify (schema ?well
                        (contains-icons ?icons)))
        (show-icon ?well ?well)
)

(defrule tickover
        (declare (salience *minimum-salience*))
=>
        (wait 5)
        (refresh 'tickover)
)
```

```
;;; -*- Mode: ART; Package: ART-USER; Base:10. -*-
;;; file: well/pattern.art

;;; Rules used to compute the pressure derivative and look for pattern
;;; in the pressure derivative, semilog and loglog curves

(defrule filter-data
        (goal (time ?well&~?? ??))
        (not (time ?well ?))
        (initial-time ?well ?i-time)
        (initial-pressure ?well ?i-pressure)
        (i-last-time ?well ?last-t)
        (interval-number ?nbr)
        (derivative ?well ?well-d)
=>
        (bind ?data (compute-average (list$ ?i-time)
                                     (list$ ?i-pressure)
                                     ?last-t
                                     ?nbr))
        (bind ?time (copy$ (cdar ?data)))
        (assert (schema ?well-d
                        (time ?time)))
        (assert (schema ?well
                        (time =(first ?data))
                        (pressure =(second ?data))
                        (last-time =(third ?data))
                        (last-pressure =(fourth ?data))))
        )

(defrule compute-p-derivative
        (time ?well ?time)
        (pressure ?well ?pressure)
        (derivative-of ?well-d ?well)
        (p-window ?l)
        (derivative-function ?func)
=>
        (assert (schema ?well-d
           (p-derivative =(funcall ?func
                                   (cdr (list$ ?pressure))
                                   (cdr (list$ ?time))
                                   ?l))))
        )

;;; A "good" pressure derivative curve should not have any positive value
;;; nor values too close to 0. The expertise proceeds only when those
;;; conditions are met.

(defrule examine-p-derivative
        (examined ?well-d positive-p-d)
        (examined ?well-d zero-p-d)
        (derivative-of ?well-d ?well)
=>
        (bind ?well-m (concat ?well 'model))
        (assert (examined ?well-d yes)
                (model ?well ?well-m))
        )

(defrule examine-positive-p-derivative
        (p-derivative ?well-d ?p-derivative)
        (test (not (member-if 'plusp (list$ ?p-derivative))))
=>
        (assert (examined ?well-d positive-p-d))
        )

(defrule examine-zero-p-derivative
        (p-derivative ?well-d ?p-derivative)
```

```
        (test (not (member-if '(lambda (x) (cond ((> x -0.01) t) (t nil)))
                              (list$ ?p-derivative))))
=>
        (assert (examined ?well-d zero-p-d))
        )

;;; Positive values are printed on the terminal and changed to -0.01.
;;; They are considered abnormal. They are given the lowest negative value.

(defrule positive-p-derivative
        (p-derivative ?well-d ?p-derivative)
        (test (member-if 'plusp (list$ ?p-derivative)))
        (time ?well-d ?time)
        (derivative-of ?well-d ?well)
=>
        (bind ?derivative (list$ ?p-derivative))
        (for p-d in ?derivative
          as tp in$ ?time
          do
         (cond ((> p-d 0)
                (fresh-line)
                (write-string "There is an abnormal positive pressure derivative in well "
                (write-string (string ?well))
                (write-string " at ")
                (write-string (write-to-string (round tp)))
                (write-string " seconds: ")
                (write-line (write-to-string p-d))
                (setf (car (member p-d ?derivative)) -0.01))))
        (modify (schema ?well-d
                        (p-derivative =?derivative)))
        )

;;; Pressure derivatives too close to 0 are changed to -0.01. This is
;;; to be coherent when the log is taken.

(defrule zero-p-derivative
        (p-derivative ?well-d ?p-derivative)
        (examined ?well-d positive-p-d)
        (test (member-if '(lambda (x) (cond ((> x -0.01) t) (t nil)))
                         (list$ ?p-derivative)))
=>
        (bind ?derivative (list$ ?p-derivative))
        (for p-d in ?derivative do
                (cond ((> p-d -0.01)
                       (setf (car (member p-d ?derivative)) -0.01))))
       (modify (schema ?well-d
                       (p-derivative =?derivative)))
        )

;;; The derivative of the pressure derivative is in fact the derivative
;;; of the log of the pressure derivative with respect to the log of time:
;;;    dlog( dP/dlog(t) )
;;;    ------------------
;;;         dlog(t)

(defrule compute-p-d-derivative
        (examined ?well-d yes)
        (time ?well-d ?time)
        (p-derivative ?well-d ?p-derivative)
        (p-d-window ?l)
        (derivative-function ?func)
=>
        (assert (schema ?well-d
           (p-d-derivative =(funcall ?func
                                     (log-of-list (list$ ?p-derivative))
                                     (list$ ?time)
```

```
                                        ?l))))
        )

;;; compute the global max and min of the pressure derivative
;;; and of its derivative

(defrule min-max-p-derivative
        (examined ?well-d yes)
        (p-derivative ?well-d ?p-derivative)
        (time ?well-d ?time)
=>
        (bind ?max (eval (append '(min) (list$ ?p-derivative))))
        (bind ?min (eval (append '(max) (list$ ?p-derivative))))
        (assert (schema ?well-d
          (max-p-derivative (=(nth$ ?time
                                    (1+ (position ?max (list$ ?p-derivative))))
                            ?max))
          (min-p-derivative (=(nth$ ?time
                                    (1+ (position ?min (list$ ?p-derivative))))
                            ?min)))
        )

(defrule min-max-p-d-derivative
        (examined ?well-d yes)
        (p-d-derivative ?well-d ?p-d-derivative)
        (time ?well-d ?time)
=>
        (bind ?max (eval (append '(min) (list$ ?p-d-derivative))))
        (bind ?min (eval (append '(max) (list$ ?p-d-derivative))))
        (assert (schema ?well-d
           (max-p-d-derivative (=(nth$ ?time
                                      (1+ (position ?max (list$ ?p-d-derivative))))
                                ?max))
           (min-p-d-derivative (=(nth$ ?time
                                      (1+ (position ?min (list$ ?p-d-derivative))))
                                ?min))))
        )

;;; looks for straight lines in the pressure derivative. A straight line
;;; is defined as a constant derivative of the pressure derivative with an
;;; admissible error and must be longer than the significant-length.

(defrule derivative-straight-lines
        (examined ?well-d yes)
        (p-derivative ?well-d ?p-derivative)
        (time ?well-d ?time)
        (p-d-derivative ?well-d ?p-d-derivative)
        (significant-length ?sig-length)
        (abs-d-error ?error)
        (rel-d-error ?rel)
=>
        (bind ?s-lines
              (straight-lines
                  (list$ ?time)
                  (list$ ?p-derivative)
                  (list$ ?p-d-derivative)
                  ?sig-length
                  (max (abs (* (car (average (list$ ?p-d-derivative))) ?rel))
                       ?error)))
        (for line in ?s-lines do
                (assert (schema ?well-d
                                (straight-line =(seq*$ line)))))
        )

;;; Two straight lines next to each other with similar slopes can be
;;; regrouped into a single straight line.
```

```
(defrule group-straight-lines
        (declare (salience 100))
   ?s1<-(straight-line ?well-d ((?ti1 ?pi1) (?tf1 ?pf1) ?sl1))
   ?s2<-(straight-line ?well-d ((?ti2&:(?ti2 > ?ti1) ?pi2) (?tf2 ?pf2) ?sl2))
        (max-p-derivative ?well-d ?max-p-d)
        (abs-d-error ?error)
        (test (and (< (abs (- ?sl1 ?sl2)) ?error)
                   (< (- (log ?ti2 10) (log ?tf1 10)) 0.25)
                   (< (abs (- ?pf1 ?pi2)) (/ (abs (nth$ ?max-p-d 2)) 10))))
=>
        (retract ?s1)
        (retract ?s2)
        (assert (straight-line ?well-d ((?ti1 ?pi1)
                                        (?tf2 ?pf2)
                                        =(/ (- (log (abs ?pf2) 10)
                                               (log (abs ?pi1) 10))
                                            (- (log ?tf2 10)
                                               (log ?ti1 10))))))
        )

;;; A straight line is computed from left to right. This rule tries
;;; to expand a straight line in the left direction. It works only for
;;; the derivative plot.

(defrule expand-straight-lines-left
        (declare (salience 100))
    ?s<-(straight-line ?well-d ((?ti ?pi) (?tf ?pf) ?sl))
        (abs-d-error ?error)
        (not (straight-line ?well-d ((?ti1 ?pi1)
                                     (?tf1&:(?tf1 < ?ti)&:(< (- (log ?ti 10)
                                                                 (log ?tf1 10))
                                                              0.2)
                                      ?pf1)
                                     ?sl1)))
        (time ?well-d ?time)
        (p-derivative ?well-d ?p-derivative)
        (p-d-derivative ?well-d ?p-d-d)
        (test (< (abs (- (nth$ ?p-d-d (1- (member$ ?ti ?time))) ?sl))
                 ?error))
=>
        (retract ?s)
        (bind ?t0 (nth$ ?time (1- (member$ ?ti ?time))))
        (bind ?p0 (nth$ ?p-derivative (1- (member$ ?ti ?time))))
        (assert (straight-line ?well-d ((?t0 ?p0)
                                        (?tf ?pf)
                                        =(/ (- (log (abs ?pf) 10)
                                               (log (abs ?p0) 10))
                                            (- (log ?tf 10)
                                               (log ?t0 10))))))
        )

;;; looks for humps in the pressure derivative. Returns the direction
;;; of the hump (valley or hill), initial time and final time.
;;; A hump is at least one log cycle long with at least one fourth of
;;; a log cycle on each side of the extremum.

(defrule humps
        (examined ?well-d yes)
        (p-d-derivative ?well-d ?p-d-derivative)
        (p-derivative ?well-d ?p-derivative)
        (time ?well-d ?time)
        (significant-length ?sig-length)
=>
        (bind ?humps (humps (list$ ?time)
                            (list$ ?p-derivative)
```

```
                                (list$ ?p-d-derivative)
                                ?sig-length))
        (for hump in ?humps do
                (assert (schema ?well-d
                                (hump =(seq*$ hump)))))
        )

;;; The extremum given by the lisp function "humps" is based on the
;;; p-d-derivative and may thus not be the exact extremum. These rules
;;; look one data point before and after and try to find a better one.

(defrule modify-hump-extremum-valley-left
        (declare (salience 100))
    ?h<-(hump ?well-d (valley (?it ?ip) (?ft ?fp) (?mt ?mp)))
        (time ?well-d ($?x ?t0 ?mt $?))
        (p-derivative ?well-d ($?y ?p-d0 ?mp $?))
        (test (and (= (length$ ?x) (length$ ?y))
                   (> ?p-d0 ?mp)))
=>
        (retract ?h)
        (assert (hump ?well-d (valley (?it ?ip) (?ft ?fp) (?t0 ?p-d0))))
        )

(defrule modify-hump-extremum-valley-right
        (declare (salience 100))
    ?h<-(hump ?well-d (valley (?it ?ip) (?ft ?fp) (?mt ?mp)))
        (time ?well-d ($?x ?mt ?t0 $?))
        (p-derivative ?well-d ($?y ?mp ?p-d0 $?))
        (test (and (= (length$ ?x) (length$ ?y))
                   (> ?p-d0 ?mp)))
=>
        (retract ?h)
        (assert (hump ?well-d (valley (?it ?ip) (?ft ?fp) (?t0 ?p-d0))))
        )

(defrule modify-hump-extremum-hill-left
        (declare (salience 100))
    ?h<-(hump ?well-d (hill (?it ?ip) (?ft ?fp) (?mt ?mp)))
        (time ?well-d ($?x ?t0 ?mt $?))
        (p-derivative ?well-d ($?y ?p-d0 ?mp $?))
        (test (and (= (length$ ?x) (length$ ?y))
                   (< ?p-d0 ?mp)))
=>
        (retract ?h)
        (assert (hump ?well-d (hill (?it ?ip) (?ft ?fp) (?t0 ?p-d0))))
        )

(defrule modify-hump-extremum-hill-right
        (declare (salience 100))
    ?h<-(hump ?well-d (hill (?it ?ip) (?ft ?fp) (?mt ?mp)))
        (time ?well-d ($?x ?mt ?t0 $?))
        (p-derivative ?well-d ($?y ?mp ?p-d0 $?))
        (test (and (= (length$ ?x) (length$ ?y))
                   (< ?p-d0 ?mp)))
=>
        (retract ?h)
        (assert (hump ?well-d (hill (?it ?ip) (?ft ?fp) (?t0 ?p-d0))))
        )

;;; A line icon is associated to each straight line and removed when
;;; the straight line is removed due to recombination or expansion.

(defrule derivative-straight-line-schema
        (logical (straight-line ?well-d ((?it ?ip) (?ft ?fp) ?slope)))
        (derivative-of ?well-d ?well)
        (x-scale ?well ?x)
```

```
        (y-scale ?well ?y)
        (last-time ?well ?last-t)
        (derivative-curve ?well-d ?)
=>
        (bind ?diff-t (- (ceiling (log ?last-t 10)) (log 0.01 10)))
        (bind ?line-name (concat ?well (gensym)))
        (bind ?xo (car (compute-coordinates ?x -2 ?diff-t (list (log ?it 10)))))
        (bind ?yo (car (compute-coordinates ?x -2 (- ?diff-t)
                                            (list (log (abs ?ip) 10)))))
        (bind ?xe (- (car (compute-coordinates ?x -2 ?diff-t
                                                (list (log ?ft 10))))
                     ?xo))
        (bind ?ye (- (car (compute-coordinates ?x -2 (- ?diff-t)
                                            (list (log (abs ?fp) 10))))
                     ?yo))
        (assert (schema ?line-name
                  (instance-of line)
                  (alu ior)
                  (translate (20 =(+ ?y 20)))
                  (thickness 2)
                  (origin (?xo ?yo))
                  (endpoint (?xe ?ye))))
        (assert (schema ?well-d
                  (derivative-icons ?line-name)))
        )

(defrule semilog-straight-line-schema
        (logical (straight-line ?well-s ((?it ?ip) (?ft ?fp) ?)))
        (semilog-of ?well-s ?well)
        (x-scale ?well ?x)
        (y-scale ?well ?y)
        (last-time ?well ?last-t)
        (pressure ?well ?pressure)
        (first-pressure ?well ?first-p)
        (last-pressure ?well ?last-p)
        (semilog-curve ?well-s ?)
=>
        (bind ?diff-t (ceiling (log ?last-t 10)))
        (bind ?diff-p (?last-p - (nth$ ?pressure 2)))
        (bind ?p0 (- ?first-p (nth$ ?pressure 2)))
        (bind ?line-name (concat ?well (gensym)))
        (bind ?xo (car (compute-coordinates ?x 0 ?diff-t (list (log ?it 10)))))
        (bind ?yo (car (compute-coordinates ?y ?p0 ?diff-p
                                            (list (- ?first-p ?ip)))))
        (bind ?xe (- (car (compute-coordinates ?x 0 ?diff-t
                                                (list (log ?ft 10))))
                     ?xo))
        (bind ?ye (- (car (compute-coordinates ?y ?p0 ?diff-p
                                                (list (- ?first-p ?fp))))
                     ?yo))
        (assert (schema ?line-name
                  (instance-of line)
                  (alu ior)
                  (translate (20 =(+ ?y 20)))
                  (thickness 2)
                  (origin (?xo ?yo))
                  (endpoint (?xe ?ye))))
        (assert (schema ?well-s
                  (semilog-icons ?line-name)))
        )

(defrule loglog-straight-line-schema
        (logical (straight-line ?well-l ((?it ?ip) (?ft ?fp) ?)))
        (loglog-of ?well-l ?well)
        (x-scale ?well ?x)
        (y-scale ?well ?y)
```

```
        (last-time ?well ?last-t)
        (first-pressure ?well ?first-p)
        (loglog-curve ?well-l ?)
=>
        (bind ?diff-t (- (ceiling (log ?last-t 10)) (log 0.01 10)))
        (bind ?line-name (concat ?well (gensym)))
        (bind ?xo (car (compute-coordinates ?x -2 ?diff-t (list (log ?it 10)))))
        (bind ?yo (car (compute-coordinates ?x -2 (- ?diff-t)
                                            (list (log (- ?first-p ?ip) 10)))))
        (bind ?xe (- (car (compute-coordinates ?x -2 ?diff-t
                                                (list (log ?ft 10))))
                     ?xo))
        (bind ?ye (- (car (compute-coordinates ?x -2 (- ?diff-t)
                                        (list (log (- ?first-p ?fp) 10))))
                     ?yo))
        (assert (schema ?line-name
                  (instance-of line)
                  (alu ior)
                  (translate (20 =(+ ?y 20)))
                  (thickness 2)
                  (origin (?xo ?yo))
                  (endpoint (?xe ?ye))))
        (assert (schema ?well-l
                  (loglog-icons ?line-name)))
        )

;;; A rectangle is associated with each hump.

(defrule hump-rectangle-schema
        (logical (hump ?well-d (? (?it ?ip) (?ft ?fp) (?mt ?mp))))
        (derivative-of ?well-d ?well)
        (x-scale ?well ?x)
        (y-scale ?well ?y)
        (last-time ?well ?last-t)
        (derivative-curve ?well-d ?)
=>
        (bind ?diff-t (- (ceiling (log ?last-t 10)) (log 0.01 10)))
        (bind ?rect-name (concat ?well (gensym)))
        (bind ?xo (car (compute-coordinates ?x -2 ?diff-t (list (log ?it 10)))))
        (bind ?yo (car (compute-coordinates ?x -2 (- ?diff-t)
                                 (log-of-list (list (min ?ip ?mp ?fp))))))
        (bind ?xe (- (car (compute-coordinates ?x -2 ?diff-t
                                  (list (log ?ft 10))))
                     ?xo))
        (bind ?ye (- (car (compute-coordinates ?x -2 (- ?diff-t)
                                 (log-of-list (list (max ?ip ?mp ?fp)))))
                     ?yo))
        (assert (schema ?rect-name
                        (instance-of rectangle)
                        (alu ior)
                        (fill clear)
                        (translate (20 =(+ ?y 20)))
                        (thickness 1)
                        (origin (?xo ?yo))
                        (endpoint (?xe ?ye))))
        (assert (schema ?well-d
                        (derivative-icons ?rect-name)))
        )

;;; This rule looks for straight lines on the semilog plot.
;;; The slope given by the lisp function is only valid for loglog or
;;; derivative plots. In this case it is recomputed.

(defrule semilog-straight-lines
        (semilog-of ?well-s ?well)
        (derivative ?well ?well-d)
```

```
        (examined ?well-d yes)
        (time ?well ?time)
        (pressure ?well ?pressure)
        (p-derivative ?well-d ?p-derivative)
        (abs-d-error ?error)
        (rel-d-error ?rel)
        (derivative-function ?func)
        (significant-length ?sig-length)
=>
        (bind ?s-lines
              (straight-lines
                  (cdr (list$ ?time))
                  (cdr (list$ ?pressure))
                  (list$ ?p-derivative)
                  ?sig-length
                  (max (abs (* (car (average (list$ ?p-derivative))) ?rel))
                       ?error)))
        (for line in ?s-lines do
                (setf (cddr line) (list (/ (- (cadar line) (cadadr line))
                                           (- (log (caar line) 10)
                                              (log (caadr line) 10)))))
                (assert (schema ?well-s
                                (straight-line =(seq*$ line)))))
        )

;;; This rule computes the derivative of the log of pressure with respect
;;; to the log of time.

(defrule log-derivative
        (loglog-of ?well-l ?well)
        (derivative ?well ?well-d)
        (examined ?well-d yes)
        (time ?well ?time)
        (pressure ?well ?pressure)
        (derivative-function ?func)
        (p-window ?l)
=>
        (assert (log-derivative ?well-l =(funcall ?func
                                                  (log-delta (list$ ?pressure))
                                                  (cdr (list$ ?time))
                                                  ?l)))
        )

;;; This rule looks for straight lines on the loglog plot.

(defrule loglog-straight-lines
        (loglog-of ?well-l ?well)
        (log-derivative ?well-l ?log-p-d)
        (derivative ?well ?well-d)
        (examined ?well-d yes)
        (time ?well ?time)
        (pressure ?well ?pressure)
        (abs-d-error ?error)
        (rel-d-error ?rel)
        (significant-length ?sig-length)
=>
        (bind ?s-lines
              (straight-lines
                  (cdr (list$ ?time))
                  (cdr (list$ ?pressure))
                  (list$ ?log-p-d)
                  ?sig-length
                  (max (abs (* (car (average (list$ ?log-p-d))) ?rel)) ?error)))
        (for line in ?s-lines do
                (assert (schema ?well-l
                                (straight-line =(seq*$ line)))))
```

```
;;; -*- Mode: ART; Package: ART-USER; Base:10. -*-
;;; file: well/model.art

;;; Rules used to extract a model by generating hypotheses and checking
;;; those hypotheses on other plots (semilog or loglog).


;;; Locates the end of early-time period by a global or local maximum

(defrule early-data-global-max
        (max-p-derivative ?well-d (?tm ?max-p-d))
        (test (< ?tm 1000))
        (derivative-of ?well-d ?well)
        (model ?well ?well-m)
=>
        (hypothesize (assert (early ?well-m ?tm)))
        )

(defrule early-data-local-max
        (max-p-derivative ?well-d (?tm&:(?tm >= 1000) ?))
        (time ?well-d ?time)
        (p-d-derivative ?well-d ($?x ?d1 ?d2 $?))
        (test (and (plusp ?d1)
                   (minusp ?d2)
                   (< (nth$ ?time (+ (length$ ?x) 2)) 10000)))
        (not (hump ?well-d (hill ? ? (?th&:((nth$ ?time (+ (length$ ?x) 2))
                                            = ?th) ?))))
        (derivative-of ?well-d ?well)
        (model ?well ?well-m)
=>
        (bind ?local-max (nth$ ?time (+ (length$ ?x) 2)))
        (hypothesize (assert (early ?well-m =?local-max)))
        )

;;; If there is a hill hump, whose maximum is different from the
;;; max-p-derivative and located before it, then, this maximum
;;; corresponds to the end of the early time period

(defrule early-data-hump
        (max-p-derivative ?well-d (?tm ?))
        (hump ?well-d (hill ? ? (?th&:(?th < ?tm) ?)))
        (derivative-of ?well-d ?well)
        (model ?well ?well-m)
=>
        (hypothesize (assert (early ?well-m ?th)))
        )

;;; If there is a negative slope at the beginning, it means that there is
;;; no early data, and that the intermediate-time period starts there.

(defrule no-early-data
        (p-d-derivative ?well-d (?fpdd&:(?fpdd < 0) $?))
        (time ?well-d (?first-t $?))
        (derivative-of ?well-d ?well)
        (model ?well ?well-m)
=>
        (hypothesize (assert (early ?well-m =?first-t)))
        )

;;; If there is a slope 1 at the beginning of the derivative curve,
;;; then there is wellbore storage.

(defrule wellbore-storage-on-derivative
        (examined ?well-d yes)
        (p-d-derivative ?well-d (?p-d1 ?p-d2 $?))
        (abs-d-error ?error)
```

```
        (test (< (abs (1- (/ (+ ?p-d1 ?p-d2) 2))) ?error))
        (derivative-of ?well-d ?well)
        (model ?well ?well-m)
=>
        (hypothesize (assert (wellbore-storage ?well-m yes)))
        )

;;; If the first two slopes at the beginning of the loglog curve are close
;;; to 1, then there is wellbore storage.

(defrule wellbore-storage-on-loglog
        (loglog-of ?well-l ?well)
        (log-derivative ?well-l (?sl1 ?sl2 $?))
        (model ?well ?well-m)
        (abs-d-error ?error)
        (test (< (abs (1- (/ (+ ?sl1 ?sl2) 2))) ?error))
=>
        (hypothesize (assert (wellbore-storage ?well-m yes)))
        )


;;; If there is a slope 1/2 at the beginning, then: fractured

(defrule fractured-on-derivative
        (examined ?well-d yes)
        (p-d-derivative ?well-d (?p-d1 ?p-d2 $?))
        (abs-d-error ?error)
        (test (< (abs (- (/ (+ ?p-d1 ?p-d2) 2) 0.5)) ?error))
        (derivative-of ?well-d ?well)
        (model ?well ?well-m)
=>
        (hypothesize (assert (fractured ?well-m yes)))
        )

;;; A 1/2 slope on loglog means fractured system

(defrule fractured-on-loglog
        (loglog-of ?well-l ?well)
        (log-derivative ?well-l (?sl1 ?sl2 $?))
        (model ?well ?well-m)
        (abs-d-error ?error)
        (test (< (abs (- (/ (+ ?sl1 ?sl2) 2) 0.5)) ?error))
=>
        (hypothesize (assert (fractured ?well-m yes)))
        )

;;; Boundary conditions: If the slope at the end is negative, hypothesize
;;; there is a pressure maintenance boundary

(defrule pressure-maintenance-boundary
        (examined ?well-d yes)
        (p-d-derivative ?well-d ($? ?pdd1 ?pdd2))
        (test (and (< ?pdd1 0)
                   (< ?pdd2 ?pdd1)))
        (derivative-of ?well-d ?well)
        (model ?well ?well-m)
=>
        (hypothesize (assert (boundary ?well-m pressure-maintenance)))
        )

;;; If the slope at the end is positive, hypothesize there is a closed
;;; system (no-flow boundary)

(defrule no-flow-boundary
        (examined ?well-d yes)
        (p-d-derivative ?well-d ($? ?pdd1 ?pdd2))
```

```
        (test (and (> ?pdd1 0)
                   (> ?pdd2 ?pdd1)))
        (derivative-of ?well-d ?well)
        (model ?well ?well-m)
=>
        (hypothesize (assert (boundary ?well-m no-flow)))
        )

;;; If, after early time, there is a doubling of the slope of the semilog
;;; curve, it means that there is a no-flow boundary

(defrule no-flow-on-semilog
        (early ?well-m ?e-time)
        (model-of ?well-m ?well)
        (semilog ?well ?well-s)
        (abs-d-error ?error)
        (straight-line ?well-s ((?it1&:(?it1 > ?e-time) ?) (?ft1 ?) ?sl1))
        (straight-line ?well-s ((?it2 ?) ? ?sl2))
        (test (and (> ?it2 ?ft1)
                   (< (abs (- ?sl2 (* 2 ?sl1))) ?error)))
=>
        (hypothesize (assert (boundary ?well-m no-flow)))
        )

;;; If there is a horizontal straight line at the end there it is an
;;; infinite system

(defrule infinite-system
        (examined ?well-d yes)
        (derivative-of ?well-d ?well)
        (last-time ?well ?last-t)
        (abs-d-error ?error)
        (significant-length ?sig-l)
        (straight-line ?well-d (? (?lt ?) ?slope))
        (test (and (< (abs ?slope) ?error)
                   (< (- (log ?last-t 10) (log ?lt 10)) ?sig-l)))
        (model ?well ?well-m)
=>
        (hypothesize (assert (boundary ?well-m infinite)))
        )

;;; If there is a valley hump, followed by either a hill hump or a
;;; horizontal straight-line then hypothesize double porosity reservoir

(defrule double-porosity
        (examined ?well-d yes)
        (abs-d-error ?error)
        (hump ?well-d (valley (?it1 ?) ? ?))
        (or (hump ?well-d (hill (?it2&:(?it2 > ?it1) ?) ? ?))
            (straight-line ?well-d ((?it3&:(?it3 > ?it1) ?)
                                    ?
                                    ?slope&:((abs ?slope) < ?error))))
        (derivative-of ?well-d ?well)
        (model ?well ?well-m)
=>
        (hypothesize (assert (reservoir ?well-m double-porosity)))
        )

;;; If there is a valley hump close to the end of the data set, then
;;; hypothezise that it is a homogeneous reservoir with a no flow boundary.

(defrule homogeneous-and-closed-system
        (examined ?well-d yes)
        (significant-length ?sig-l)
        (derivative-of ?well-d ?well)
        (last-time ?well ?last-t)
```

```
        (hump ?well-d (valley ? (?lt ?) ?))
        (test (< (- (log ?last-t 10) (log ?lt 10)) ?sig-l))
        (model ?well ?well-m)
=>
        (hypothesize (assert (reservoir ?well-m homogeneous)
                             (boundary ?well-m no-flow)))
        )

;;; If there is a hill hump followed by a straight line until the end
;;; of the data set, then hypothesize that it is a homogeneous reservoir
;;; with an infinite boundary

(defrule homogeneous-and-infinite-system
        (examined ?well-d yes)
        (significant-length ?sig-l)
        (derivative-of ?well-d ?well)
        (last-time ?well ?last-t)
        (abs-d-error ?error)
        (hump ?well-d (hill ? (?hump-lt ?) ?))
        (straight-line ?well-d ((?line-it ?) (?line-lt ?) ?slope))
        (test (and (< ?hump-lt ?line-it)
                   (< (- (log ?line-it 10) (log ?hump-lt 10)) 0.25)
                   (< (abs ?slope) ?error)
                   (< (- (log ?last-t 10) (log ?line-lt 10)) ?sig-l)))
        (model ?well ?well-m)
=>
        (hypothesize (assert (reservoir ?well-m homogeneous)
                             (boundary ?well-m infinite)))
        )

;;; If there is a hill hump and no valley hump, then hypothesize
;;; homogeneous

(defrule homogeneous
        (hump ?well-d (hill ? ? (?th ?)))
        (derivative-of ?well-d ?well)
        (model ?well ?well-m)
        (early ?well-m ?th)
        (not (hump ?well-d (valley ? ? ?)))
=>
        (hypothesize (assert (reservoir ?well-m homogeneous)))
        )

;;; These rules merge viewpoints. They aggregate all possibilities for every
;;; possible combination of facts in the well-model schemata. It also prints
;;; all possible models for each well

(defrule find-model-with-wellbore-storage
        (early ?well-m ?)
        (wellbore-storage ?well-m yes)
        (reservoir ?well-m ?type)
        (boundary ?well-m ?bound)
        (model-of ?well-m ?well)
=>
        (fresh-line)
        (write-string "A possible model for well ")
        (write-string (string ?well))
        (write-string " is a ")
        (write-string (string ?type))
        (write-string " reservoir with wellbore storage and a ")
        (write-string (string ?bound))
        (write-string   " boundary.")
        )

(defrule find-fractured-model
        (early ?well-m ?)
```

```
        (fractured ?well-m yes)
        (reservoir ?well-m ?type)
        (boundary ?well-m ?bound)
        (model-of ?well-m ?well)
=>
        (fresh-line)
        (write-string "A possible model for well ")
        (write-string (string ?well))
        (write-string " is a ")
        (write-string (string ?type))
        (write-string " reservoir with a fractured well and a ")
        (write-string (string ?bound))
        (write-string   " boundary.")
        )

(defrule find-model
        (early ?well-m ?)
        (reservoir ?well-m ?type)
        (boundary ?well-m ?bound)
        (not (or (wellbore-storage ?well-m yes)
                 (fractured ?well-m yes)))
        (model-of ?well-m ?well)
=>
        (fresh-line)
        (write-string "A possible model for well ")
        (write-string (string ?well))
        (write-string " is a ")
        (wrte-string (string ?type))
        (write-string " reservoir with a ")
        (write-string (string ?bound))
        (write-string   " boundary.")
        )
```

```
;;; -*- Mode: LISP; Package: ART-USER; Base:10. -*-
;;; file: well/well.lisp

;;; Lisp functions used principally for computing derivatives and
;;; looking for patterns, like straight-lines and humps.

(defun seconds (data)
        (+ (* 3600 (first data)) (* 60 (second data)) (third data)))

(defun concat (name suffix)
        (make-symbol (concatenate 'string (string name) "-" (string suffix))))

(defun read-from-file (well)
        (let* ((file (concatenate 'string "~marc/data/" (string well)))
               (stream (open file))
               (units (read stream))
               (time-unit (car units))
               (pressure-unit (cadr units))
               (first-data (read stream))
               (initial-time (* time-unit (seconds first-data)))
               (initial-pressure (* pressure-unit (fourth first-data)))
               (last-time 0)
               (last-pressure initial-pressure)
               (x-pts (list 0))
               (y-pts (list initial-pressure)))
              (do ((data (read stream)
                         (read stream nil)))
                  ((not data))
                  (nconc x-pts (list (setq last-time
                                           (- (* time-unit (seconds data))
                                              initial-time))))
                  (nconc y-pts (list (setq last-pressure (* pressure-unit
                                                            (fourth data))))))
              (close stream)
              (list x-pts y-pts last-time initial-pressure last-pressure)))

(defun compute-coordinates (scale first diff data-set)
        (mapcar '(lambda (data) (round (* ,scale (/ (- data ,first) ,diff))))
                data-set))

(defun delta (data-set)
        (mapcar '(lambda (data) (- ,(car data-set) data)) (cdr data-set)))

(defun log-of-list (data-set)
        (mapcar '(lambda (data) (log (abs data) 10)) data-set))

(defun log-delta (data-set)
        (mapcar '(lambda (data) (log (- ,(car data-set) data) 10))
                (cdr data-set)))

(defun compute-average (i-time i-pressure last-time int-number)
        (let ((time '(0))
              (pressure (list (car i-pressure)))
              (factor (/ (log last-time 10) int-number))
              (t-ave ())
              (p-ave ()))
             (setq i-time (cdr i-time))
             (setq i-pressure (cdr i-pressure))
             (dotimes (i0 int-number (list time
                                           pressure
                                           (* 1.0 (car (average t-ave)))
                                           (car (average p-ave))))
                 (let ((i (1+ i0)))
                      (setq t-ave ())
                      (setq p-ave ())
                      (do ((t-data (car i-time)
```

```
                                    (car (setq i-time (cdr i-time))))
                              (p-data (car i-pressure)
                                    (car (setq i-pressure (cdr i-pressure)))))
                             ((or (not t-data)
                                  (> t-data (+ (expt 10 (* i factor)) 0.0000002)))
                              (setq time (append time (average t-ave)))
                              (setq pressure (append pressure (average p-ave))))
                             (setq t-ave (append t-ave (list t-data)))
                             (setq p-ave (append p-ave (list p-data)))))))


(defun average (list)
        (let ((l (list-length list)))
             (cond ((= l 0) ())
                   (t (list (/ (eval (append '(+) list)) l))))))

(defun w-ave (x y w1 w2)
        (/ (+ (* w1 x) (* w2 y)) (+ w1 w2)))

(defun derivative-log-time-B-half (pressure time l)
   (mapcar `(lambda (p0 t0)
                 (let* ((x (position t0 ',time))
                        (ix (max (- x ,l) 0))
                        (fx (min (+ x ,l) (1- (length ',time))))
                        (ip (nth ix ',pressure))
                        (fp (nth fx ',pressure))
                        (it (nth ix ',time))
                        (ft (nth fx ',time)))
                       (cond ((= x ix) (/ (* t0 (- fp p0)) (- ft t0)))
                             ((= x fx) (/ (* t0 (- p0 ip)) (- t0 it)))
                             (t (* (/ t0 2)
                                   (+ (/ (- p0 ip) (- t0 it))
                                      (/ (- fp p0) (- ft t0)))))))))
           pressure time))

(defun derivative-log-time-B (pressure time l)
   (mapcar `(lambda (p0 t0)
                 (let* ((x (position t0 ',time))
                        (ix (max (- x ,l) 0))
                        (fx (min (+ x ,l) (1- (length ',time))))
                        (ip (nth ix ',pressure))
                        (fp (nth fx ',pressure))
                        (it (nth ix ',time))
                        (ft (nth fx ',time)))
                       (cond ((= x ix) (/ (* t0 (- fp p0)) (- ft t0)))
                             ((= x fx) (/ (* t0 (- p0 ip)) (- t0 it)))
                             (t (* t0 (w-ave (/ (- p0 ip) (- t0 it))
                                             (/ (- fp p0) (- ft t0))
                                             (- ft t0)
                                             (- t0 it)))))))
           pressure time))

(defun derivative-logtime (pressure time l)
   (mapcar `(lambda (p0 t0)
                 (let* ((long (length ',time))
                        (x (position t0 ',time))
                        (ix (cond ((= x (1- long)) (- long 2))
                                  ((= x (- long 2)) (- long 3))
                                  ((= x (- long 3)) (- long 5))
                                  (t (max (- x ,l) 0))))
                        (fx (cond ((= x 0) 1)
                                  ((= x 1) 2)
                                  ((= x 2) 4)
                                  (t (min (+ x ,l) (1- long)))))
                        (ip (nth ix ',pressure))
                        (fp (nth fx ',pressure))
```

```
                    (ot (log t0 10))
                    (it (log (nth ix ',time) 10))
                    (ft (log (nth fx ',time) 10)))
                  (cond ((= x ix) (/ (- fp p0) (- ft ot)))
                        ((= x fx) (/ (- p0 ip) (- ot it)))
                        (t (w-ave (/ (- p0 ip) (- ot it))
                                  (/ (- fp p0) (- ft ot))
                                  (- ft ot)
                                  (- ot it))))))
          pressure time))


(defun straight-lines (time p-derivative p-d-derivative sig-length error)
  (let ((s-lines ())
        (ti 0)
        (tf 0)
        (pdi 0)
        (pdf 0)
        (ave 0))
       (do ((first (car p-d-derivative)
                   (car p-d-derivative)))
           ((not first) s-lines)
           (setq ti (car time))
           (setq pdi (car p-derivative))
           (setq time (cdr time))
           (setq p-derivative (cdr p-derivative))
           (setq s-line
             (do* ((current (car (setq p-d-derivative (cdr p-d-derivative)))
                            (car (setq p-d-derivative (cdr p-d-derivative))))
                   (line (list first) line)
                   (ave first (car (average line))))
                  ((or (not current)
                       (> (abs (- current ave)) error)
                       (> (abs (- ave first)) error))
                   line)
                  (setq line (append line (list current)))
                  (setq tf (car time))
                  (setq pdf (car p-derivative))
                  (setq time (cdr time))
                  (setq p-derivative (cdr p-derivative))))
           (setq s-lines (append s-lines
             (cond ((and (plusp tf)
                         (>= (- (log tf 10) (log ti 10)) sig-length))
                    (list (list (list ti pdi)
                                (list tf pdf)
                                (/ (- (log (abs pdf) 10)
                                      (log (abs pdi) 10))
                                   (- (log tf 10)
                                      (log ti 10))))))
                   (t ()))))))))

(defun humps (time p-derivative p-d-derivative sig-hump)
  (let ((humps ())
        (otime ())
        (op-derivative ())
        (op-d-derivative ())
        (ti ())
        (tf ())
        (top ()))
    (do ((first (car p-d-derivative)
                (car p-d-derivative)))
        ((not first) humps)
        (setq ti (list (car time) (car p-derivative)))
        (setq top ())
        (setq humps (append humps
            (do* ((current first (car p-d-derivative))
```

```
 (after-top 0 (if top (car time) 0))
 (next (car (setq p-d-derivative (cdr p-d-derivative)))
       (car (setq p-d-derivative (cdr p-d-derivative))))
 (bef-noise
    current
    (if (or (and (plusp first)
                 (< current bef-noise)
                 (if next (< next current) t))
            (and (minusp first)
                 (> current bef-noise)
                 (if next (> next current) t)))
        current bef-noise))
 (noise (list (car time) (car time))
        (if (= current bef-noise)
            (list (car time) (car time))
            (list (car noise) (car time)))))
((or (not next)
     (>= (- (log (cadr noise) 10) (log (car noise) 10))
         sig-hump))
 (cond (next (setq time otime)
             (setq p-derivative op-derivative)
             (setq p-d-derivative op-d-derivative)))
 (cond ((and (plusp after-top)
             (>= (- (log after-top 10) (log (car ti) 10))
                 (* 3 sig-hump))
             top
             (>= (- (log (car top) 10) (log (car ti) 10))
                 sig-hump)
             (>= (- (log after-top 10) (log (car top) 10))
                 sig-hump))
        (list (list (cond ((plusp first) 'hill)
                          (t 'valley))
                    ti tf top)))
       (t ())))
(if (and (= after-top 0)
         (/= (signum first) (signum current)))
    (setq top (list (car time) (car p-derivative))))
(setq p-derivative (cdr p-derivative))
(setq time (cdr time))
(cond ((and (<= (- (log (cadr noise) 10) (log (car noise) 10))
                (/ sig-hump 2))
            (or (zerop after-top)
                (<= (- (log after-top 10) (log (car top) 10))
                    sig-hump)))
       (setq otime time)
       (setq op-derivative p-derivative)
       (setq op-d-derivative p-d-derivative)))
(if (= (car noise) (cadr noise))
    (setq tf (list (car time) (car p-derivative))))
))))))
```

LAWRENCE BERKELEY LABORATORY
TECHNICAL INFORMATION DEPARTMENT
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720