Task Interaction and Control System (TICS)

Mark Grossman
M.S., University of California, San Diego, 1984
B.A., State University of New York, Stony Brook, 1972

A dissertation submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

June, 1987

The dissertation "Task Interaction and Control System (TICS)", by Mark Grossman, has been examined and approved by the following Examination Committee:

David E. Maier, Thesis Advisor
Associate Professor

Lougenia Anderson
Principal Scientist, Tektronix, Inc.

Robert G. Babb II
Associate Professor

Richard Hamlet
Professor

Dedication

To my parents for giving me the ability,

my son for providing the inspiration,

and my advisor for his time and ideas.

Table of Contents

List of Figures

Abstract

Task Interaction and Control System (TICS)

Mark Grossman, Ph.D.
Oregon Graduate Center, 1987

Supervising Professor: David E. Maier

Task Interaction and Control System (TICS) models computer-assisted problem-solving as the decomposition of a problem into subtasks. TICS provides a declarative and executable specification of such a model through the use of Horn clause logic. The logic clearly expresses the assumptions and rules that control the composition and interaction of the predicates that solve the subtasks. The power of logic programming to represent a hierarchical search for task solutions is augmented in TICS by evaluable predicates. Evaluable predicates invoke external procedures that execute as concurrent processes and whose internals are hidden from TICS.

Logic provides a rich framework to specify a space of ways to solve a problem. This approach differs from the use of non-declarative codes and scripts to manage tasks, which usually permit only a subset of the possible solutions. TICS promotes user-directed exploration. Logic does not inflict artificial orderings or dependencies among subtasks that are not present in the problem domain itself. The specific order to solve the subtasks is not dictated, the user's search being limited only by those constraints inherent in the task and not by the rigidity of a computer program.

TICS encourages trial-and-error problem-solving by carefully tracking true subtask dependencies, and minimally undoing previous work through the use of intelligent backtracking. TICS supports multiple techniques to cope with the need to undo side-effects.

A prototype system was implemented in C++. Using this system to solve problems we noted that freedom from artificial constraints sometimes created a burden of choices. TICS was enhanced to provide a flexible means to limit the number of decisions that an end-user must consider. To guide decisions, a natural language format and menu facility were incorporated to communicate with the user. The versatility of the TICS framework is illustrated through example problems that have been implemented on our prototype system. We conclude by discussing our experience with TICS and topics for further research.

Chapter 1

Introduction

Etiquette is a formal code of behavior set up to regulate human interaction. A person who deviates from the prescribed path may be frowned upon and considered to be in error. Etiquette provides a way to implement an interactive system with a minimum of friction. However, there is a price. This highly structured interaction, by its rigid nature, can be exasperating in its intolerance of deviation. The system tends to suppress individuality, creativity, originality, and exploration. In short, such a system, by its intolerance of deviation, has an element of frustration and boredom, and is not much fun. Most people would categorize their interactions with computers in just this way. A highly constrained interactive system, however easy to implement, limits the user's repertoire of approaches. The ability to effectively solve problems with the system is greatly constrained.

To relax the rules we must keep our minds open and realize that deviation from expectations can be positive. Not only should systems be designed to be tolerant of errors, but the system, ideally, should encourage the experimentation that can result in them. Deviations from behavior patterns, like biological mutations, allow for the random incorporation of changes that might prove useful in current and future circumstances. These deviations provide for evolution. Because of them users become better adapted to their environment.

In solving a complex task, a user cannot predict the ramifications of every decision. A computer system should not force the user to work out all of the implications and interplay between decisions before entering them, but should help the user explore. Users should be able to change their minds without getting browbeaten by the system. Serendipity is just another name for an error that proved useful. A system creator should say, "Let there be errors." Providing a framework for the specification and implementation of such user-oriented systems is the objective of our Task Interaction and Control System (TICS).

Most people fret not for philosophers who are starving for lack of utensils and common sense, and most people have two, not eight, queens on their chess boards. They have no desire to repeatedly spend an unending amount of time streaming an infinite list of numbers into an equally large number of prime sieves. Problems of resource allocation, optimal search strategies and elegant abstract algorithms lie in the domain of computer specialists. The *real-world* problems faced by people are often of a different nature. Solutions are developed in the context of the user's *world* of facts and rules. This *world* defines what constitutes an acceptable solution and ensures that constraints imposed by reality are not violated. A typical individual solves a complex problem by decomposing the task into more manageable subtasks, each of which focuses on one aspect of the overall problem. Difficulties often arise because there are inter-dependencies between the subtasks that may prevent a linear ordering of their solutions. The solution of one subtask may depend upon and affect the solution of other subtasks. It may be impossible or intractable to completely

solve each subtask before proceeding to solve the next.

1.1. Illustrative Example

We are all at least somewhat familiar with the process of buying a car, a task described as a pleasure by our friends in the advertising business. An individual is limited by a number of inherent constraints while solving this problem. Some define what is an acceptable solution, e.g., the required delivery date of the vehicle. Other constraints define necessary temporal orderings reflecting the cause-and-effect relationships being modeled. For example, one cannot determine the price of a new vehicle until that vehicle's model is specified.

Within these essential constraints a number of solutions are possible. A person will choose solutions based upon his or her particular circumstances. In a simple situation one may need only to select the new car's style. For this example we will examine the more usual and complex case requiring decomposing the task into the the following subtasks: disposing of an old vehicle, choosing the new vehicle, and arranging for financing. Choosing a model is further decomposed into subtasks to specify the body style and color. A purchaser may quickly solve the task by using the package solution provided by a helpful car salesperson. This solution might consist of using the old car as a trade-in and financing the new car through the dealer. Most people prefer to customize their purchase, deriving a better solution for their particular situation.

Purchasing a vehicle does not inherently require that any one subtask be started before the other. One person may wish to establish the value of their old

vehicle before considering options for the new vehicle; another may first wish to determine the amount and cost of available financing. It is important to note that different people may not only handle the tradeoffs among solutions differently, but also may prefer to reach an overall solution via different orderings of subtasks.

Regardless of what direction the solution initially takes, decisions made while solving each subtask will affect other, possibly completed, subtasks. For instance, choosing a certain model could conflict with the required delivery date because such a vehicle is not in stock and would have to be special ordered. Information derived while solving a subtask is cooperatively used by other subtasks, even before that subtask is completed. Once a person selects the body style of the new vehicle, the choice of colors offered is restricted to those available for that model. Alternatively, if a color is first selected, then the range of body styles must be constrained to allow only those that come in the selected color.

If a constraint is violated by an action, then a person must eventually undo a previously solved subtask to resolve the problem. If selecting a certain model causes the cost of the vehicle to exceed the amount of money available, an individual must choose from among the alternatives to resatisfy the constraint. For example, one could dispose of the old vehicle privately to raise more money, or one could select a different model, to lower the price.

There are other reasons besides resolving constraint violations for a person wanting to undo some previous action. A user cannot be expected to have complete knowledge of the interrelationships of a complex system. An iterative approach may

be required by the user to find a satisfactory solution. The ability to change answers provides a person with the capability to solve a problem by trying out different choices and exploring their ramifications. One might want tentatively to change the selected vehicle's color to learn the effect on the delivery date. Individuals sometimes use existing prototypical solutions. A person might want to start with a salesman's initial proposal and modify it, rather than to start from scratch. Input errors are sometimes made and facts are sometimes altered. When a salesperson believes that a purchase will not be made because of monetary considerations, the price of the new vehicle may be reduced or the amount offered for a trade-in increased.

It should be as easy as possible to undo previous actions. Work done to solve parts of a problem that are independent of a modification should not be lost. The sequence in which subtasks are solved must not be confused with causality. Regardless of the order of solving a new vehicle's color and financing sub-problems, a change in vehicle color that does not affect the car's price should not cause the financing sub-problem to be undone. In some cases, subtask dependencies will cause a modification to one subtask to affect other solved parts of the problem. Ideally, a person should be aware of these impacts. For example, if an individual decides to change the previously selected style, he or she should be informed if this will affect the price or delivery date.

When solving a task it is often necessary to communicate with agents outside the task's environment and direct control. These actions can cause changes that are

not reflected in the state of the task. We term these changes side-effects. When undoing a subtask that has caused a side-effect, corrective action, when possible, should be taken. For example, when a person solves the financing subtask certain actions might be initiated, such as sending a letter to the finance company requesting a loan for a new vehicle. If the person later decided to cancel the purchase, a new letter should be generated to the finance company to attempt to undo the effects of the previous action. Some actions cannot be undone; it is not possible to restore missiles to their silos. These sorts of side-effects should therefore be delayed as long as possible to allow for changes of heart.

Some information should be retained even if the parent subtask is undone and another approach tried. If the financing subtask must be redone, due to a slight modification in the amount to be borrowed, a person should not have to once again fill out all the parts of a long and detailed loan form. Information, such as a person's name and address, should not have to be entered multiple times.

1.2. Summary of Goals

We want TICS to support the features of *real-world* problem solving outlined above. A system that claims to be user-oriented needs to address the points that were illustrated in the previous section. Such a system should:

(1) Provide a clear and concise way to specify what constitutes an acceptable solution to a task. This specification should:

A. Decompose a task into meaningful subtasks.

B. Permit information to be supplied from any one of a number of sources.

C. Allow for the use of multiple ways to solve a task or subtask.

(2) Permit subtasks to be solved in any order consistent with the inherent nature of the task. The temporal order of solutions should not be confused with causality.

(3) Have the capability for subtasks to cooperate to solve inter-dependent subproblems.

(4) Recognize independent subtasks and track dependencies to provide a general and intelligent undo facility that allows the system or user to modify previous answers with a minimum amount of lost effort.

(5) Support communication with agents outside the system, with the capability to handle, as well as possible, side-effects.

(6) Allow some information to persist even if the procedure that generated the information is undone. Such persistent data can be used to establish alternative solutions to a subtask.

The structure of a task and the environment in which it is to be solved needs to be developed by a designer. A designer-oriented problem solving system should:

(7) Include facilities to hierarchically modularize a complex system to allow different levels of abstraction and details.

(8) Have the flexibility to develop a system using a wide variety of tools and environments.

(9) Allow for the easy re-use of existing solutions.

(10) Enhance development of software for diverse groups of end-users by incorporating the ability to separate the user-interface processes from application processes.

(11) Provide a methodology to distribute the solving of subtasks to multiple processors. This methodology should include a means for these subtasks to communicate and synchronize.

1.3. Computer Algorithms as Task Solvers

Computer science has always been concerned with problem solving and therefore with the nature of problems. Efficient algorithms exists to solve many types of tasks, for example, sorting a list of numbers. Some complicated tasks have a nature that allows them to be decomposed into subtasks in such a way that optimally solving the subtasks implies a global optimum solution. An example of this is the *Minimum-Cost Alphabetic Tree* problem [Hu 82]. The key here is that although there are an exponential number of decompositions, there are only a polynomial number of non-interacting sub-problems. A general method for solving these types of tasks is provided by *Dynamic Programming* [Hu 82].

All the tasks mentioned in the last paragraph have a nature that permits their search space of possibilities to be analyzed in a reasonable amount of time in order to determine an optimal solution. However, many other problems are intractable because solving any sub-part of the problem affects many other sub-parts. If one

sub-part of the problem is solved, other sub-parts must be re-examined in order to ensure that local decisions are not preventing the system from achieving a better global solution. This inter-dependency causes a combinatorial explosion of possibilities that must be examined to solve the task optimally; there are more solutions than can be generated and tested in a reasonable amount of time. The *Bin-Packing* problem exemplifies this intractable nature [Hu 82]. Recent work has lead to a general heuristic approach for solving such problems. This method called *Simulated Annealing* is described in detail by Kirkpatrick, Gelatt and Vecchi [Kirkpatrick, Gelatt and Vecchi 83]. Simulated Annealing involves adding noise, i.e. variation, to the sub-solutions of a task. The result, if done correctly, is to move the system out of states of local optimality in order to achieve a better global solution. The amount of noise is then reduced with the result that a near perfect solution can be determined in an acceptable amount of time.

If one had to examine all the possibilities required to purchase a vehicle before making a decision then one might prefer to walk. The approach most people take is to make good initial selections for each of the subtasks and spend a reasonable amount of time slightly modifying them, i.e., introducing noise, and noting the overall effects. Each change may produce positive and negative effects. The results are examined to determine whether the total solution has been improved by the tradeoffs, and this is used to guide further exploration.

Chapter 2

TICS' Approach

Solving complex problems requires the ability to model and manipulate synergistic systems. TICS provides a framework to develop such interactive human-computer systems and to incorporate in them the most desirable features of effective problem solving, illustrated and summarized in the previous chapter.

2.1. Overview of TICS

In this section the important features of TICS are illustrated. The first part of our discussion concerns the advantages of using Horn clause logic to model the decomposition of problems. Next, we discuss the extensions to Horn clause logic required to achieve the goals described in the previous chapter. The motivation for augmenting logic programming with evaluable predicates is presented. TICS' resolution mechanism, which allows subtasks to interact with each other and which provides user- and system-initiated undo is described. Techniques for dealing with the problems of side-effects and for preserving and re-using information are discussed. We conclude with an example of how our prototype TICS implementation provided a flexible and forgiving interface for a Pascal program written by Knuth [Bentley and Knuth 86].

2.2. Horn Clause Logic

TICS uses Horn clause logic to specify tasks. Some cognitive psychologists feel that the Horn clause subset of first-order logic compares favorably with other formalisms as a basis for information processing models of human problem-solving [Nilsson 80]. The reader is referred to Clocksin and Mellish [Clocksin and Mellish 84] for the syntactic conventions used by TICS and for a detailed technical description of Horn clauses and their relationship to logic in general. A major advantage of a Horn clause specification is that it is runnable. A task can be formulated as a goal, i.e., a headless clause, that can be proved, using a strategy based upon resolution. Horn clauses allow us to declaratively specify facts and rules succinctly to describe what constitutes a solution. The problem-reduction strategy for Horn clauses is identical with the procedural interpretation of Horn clauses and naturally represents the decomposition of a task into meaningful subtasks, the first part of Goal (1). The procedural interpretation is described by Kowalski as follows (with clause syntax modified to fit our usage):

> An implication of the form
>
> A :- B_1 , ... , B_n.
>
> is interpreted as a *procedure* which reduces problems of the form A to subproblems B_1 and ... B_n. Each of the subproblems B_i in turn is interpreted as a *procedure call* to other implications. Assertions A. are interpreted as procedures A :- . which solve problems directly by reducing them to an empty collection of subproblems.
>
> To apply a procedure to a procedure call, it may be necessary to instantiate variables, to make the procedure call identical to the conclusion of the procedure. Instantiating variables in the procedure can be regarded as transmitting input from the procedure call to the procedure. Instantiating variables in the procedure call can be regarded as transmitting output from the procedure to the procedure call and to all other procedure calls with which it shares variables. [Kowalski 82]

The following clause illustrates one simple decomposition of the task, i.e. procedure, of purchasing a car:

```
buy_car(
  tradein(TMethod, TModel, TValue),
  new_car(NModel, NPrice, NDelivery),
  finance(FinanceMethod, MonthlyPayments))   :-
     do_tradein(TMethod, TModel, TValue),
     select_new(NModel, NPrice, NDelivery),
     arrange_finance(TValue, NPrice, FinanceMethod,
          MonthlyPayments).
```

Each use of a procedure establishes a local binding environment with new instances of the logical variables. The local environment contains a subset of the data that defines the overall state of the system. In the `buy_car` clause the procedure used to solve the `arrange_finance` subtask is specified to contain, in its environment, variables that contain information regarding the value of the old vehicle, cost of the new vehicle, financing method and the required monthly payments for the loan. Each procedure can view and modify its local binding environment and thereby examine and change its specified part of the system's state. Procedures can exchange information through communication channels, i.e., variables in each of their local environments that are constrained, via unification, to contain the same value. The procedure used to solve the `select_new` subtask could instantiate its local variable that is unified to the `NPrice` variable. This instantiated value would then be accessible to the procedure used to solve the `arrange_finance` subtask.

Logic variables can allow indeterminism as to which subtask supplies a value for a variable. This permits information to be supplied from any one of a number of

sources, the second part of Goal (1). For example, a person might wish to determine how expensive a vehicle he or she could purchase with a specific monthly payment. One could invoke the buy_car task instantiating the MonthlyPayments variable to 100 dollars as follows:

```
?- buy_car(tradein(TMethod, TModel, TValue),
      new_car(NModel, NPrice, NDelivery),
      finance(FinanceMethod, 100)).
```

Alternatively, a person could provide enough information about the rest of the problem so that the arrange_financing subtask could calculate and instantiate MonthlyPayments.

Another feature of logic, non-determinism, allows more than one clause, i.e., method of solution, to be applicable to a given procedure call. Thus, multiple ways can be specified to solve a subtask, the final requirement of Goal (1). In our car example, we might add to our problem specification another clause to provide a default prototypical solution for the case where there is no trade-in vehicle and dealer financing is to be used.

```
buy_car(
  tradein(none, none, 0),
  new_car(NModel, NPrice, NDelivery),
  finance(dealer, MonthlyPayments)) :-
     select_new(NModel, NPrice, NDelivery),
     arrange_finance(0, NPrice, dealer,
          MonthlyPayments).
```

Logically, the collection of procedure calls listed in the method used to solve a task, i.e., the body of the selected clause, can be executed in any order and even in parallel. Some systems, e.g., Prolog, impose an ordering for the sake of efficiency and

simplicity. TICS uses the more flexible method of resolution theorem proving based on *plan-based deduction* [Cox and Pietrzykowski 81] [Forsythe and Matwin 84] [Matwin and Pietrzykowski 85]. No ordering is therefore imposed by the specification upon the solving of subtasks. Subtasks can be solved in any order consistent with the inherent nature of the task. (Goal (2)). Thus, if the first `buy_car` clause was used to solve the task, one could work on `do_tradein`, `select_new` or `arrange_finance` in any order or in parallel. Although the utility of plan-based deduction for theorem proving is unclear, the performance is adequate for TICS because of the incorporation of evaluable predicates, as described below.

2.3. Extensions to Logic

Evaluable Predicates

People typically decompose problems until they get to a subtask with a deterministic, functional solution. Subtasks that have a straightforward or algorithmic solution require no further decomposition. The internal structure of such procedures is not part of the decomposition of the problem and is, therefore, of limited interest to the problem solver. TICS provides for *evaluable predicates*, i.e., subtasks that are solved by external procedures. These procedures are implemented as processes that can incorporate large grain functionality and whose internals are hidden from TICS. Logically, an evaluable predicate's procedure can be viewed as a dynamic fact generator. Such predicates can be specified to be either of type *external* or *external generator*, the difference being that the latter type process can be re-invoked, by backtracking, to generate alternative facts. TICS' evaluable predicates eliminate the

need to incorporate the bottom level of detail in the logic. Thus, TICS' large overhead, required to track dependencies, can be supported.

A method, based upon the technique of Gray, Moffat and du Boulay [Gray, Moffat and Boulay 85] [Moffat and Gray 86], is provided by TICS to handle external objects safely, i.e., typed data from other languages, within a type-free logic environment. We discuss this method further in Chapter 4.

And-parallelism

To provide the capability for subtasks to cooperate to solve inter-dependent sub-problems, Goal (3), we modified plan-based deduction to support *and-parallelism.* Multiple procedures can be executed by external processes running concurrently with each other (and with TICS). A TICS' procedure can receive and send information via logical variables throughout its execution, not just at initiation and completion. This ability requires care with bindings of logical variables. Therefore, TICS maintains a partial ordering of process dependencies reflecting causality as described by Lamport [Lamport 78].

Procedures to solve subtasks are temporally ordered only by the data they require, i.e., the inherent constraints of the problem. If a procedure requires data that is not yet available it must wait. TICS provides a notification mechanism that allows external procedures to synchronize with each other. A procedure may wait until TICS notifies it that a variable's binding has changed. For example, while the `arrange_finance` subtask may be started at any time, it must wait until TICS notifies it that the values for `NPrice` and `TValue` have been supplied by

select_new and do_tradein, respectively, before computing MonthlyPayments.

In TICS subtasks can cooperate. The partial solution of one subtask can constrain another. When a person selects the style of the new vehicle, this information can be used to restrict the choice of colors to only those available in that style. Alternatively, if a color is first selected, then the range of styles can be limited to only those that come in the selected color. The following procedure is an example of specifying subtasks with appropriate communication channels:

```
select_new(NModel, NPrice, NDelivery) :-
      get_style(NStyle, Color), get_color(NStyle, Color),
      find_model(NStyle, Color, NModel).
```

Undo

As Donald Norman has stated, "Error is the natural result of a person attempting to do a task". Therefore, it should be as easy as possible to undo previous actions. Work done to solve parts of a problem that are independent of a modification should not be lost. A system must not confuse the sequence in which subtasks are solved with causality. For example, regardless of the order in which a new vehicle's color and financing subtasks are solved, a change in vehicle color that does not affect the car's price should not cause the financing subtask to be undone.

Logic is a good framework for tracking dependencies. TICS' extended version of plan-based deduction maintains information about the history of the resolution, the unification constraints between variables, and the causal relationships between

binding environments. With this information TICS can recognize independent subtasks and track dependencies to provide a general and intelligent undo facility that allows the system or user to modify previous answers with a minimum amount of effort, (Goal (4)). Intelligent backtracking detects and acts upon the exact source of failure as opposed to exhaustive, blind backtracking, which acts upon the most recent procedures first. TICS allows alternative solution paths to be tried and errors to be corrected with only solutions to those subtasks affected by the change needing to be redone. In our car example, if the value for `NPrice`, supplied by the `select_new` procedure, was undone after `arrange_finance` had used its value, then `arrange_finance` would need to be redone. However, even if `do_tradein` had been solved after `select_new` completed, `do_tradein` would be unaffected because `do_tradein` and `select_new` do not share variables. In addition, if `arrange_finance` had not yet examined the undone value for `NPrice`, there would be no causal dependency and thus, `arrange_finance` would not need to be redone.

Side-effects

A side-effect is an effect not contained and reflected in the state, i.e., logical variable bindings, of the system. Procedures that cause no side-effects are undone via TICS' backtracking mechanism by removing the logical variable bindings caused by them. However, the execution of certain procedures can cause side-effects that are not contained and reflected in TICS' logical variables. A common problem is the side-effect caused by information communicated beyond the boundary of the system

[Archer, Conway and Schneider 84]. The issue of undoing side-effects is difficult, but must be addressed in any real system. TICS provides two different ways to address this issue. An evaluable predicate can be specified to have paired with its normal procedure either a *delayed* or *inverse* procedure.

Some subtasks have side-effects that can be delayed until the user commits to an answer. Such side-effects can be incorporated into delayed procedures. These procedures are executed at the completion of the deduction only if their invoking subtask is still valid. If the subtask was withdrawn during backtracking, the delayed procedure is never executed and no side-effect occurs. In our previous car-buying program we could specify that a delayed procedure be associated with the `arrange_finance` subtask. When the user commits to a solution, thus completing the buy_car task, the procedure associated with `arrange_finance` is executed. This procedure would cause the side-effect of mailing a loan request to the financing agency.

Sometimes a subtask requires that side-effects not be delayed. Thus, the normal procedure associated with such a subtask would itself cause the side-effect to occur. Such a subtask could specify an inverse procedure that is executed when the specifying subtask is withdrawn. The inverse procedure is used to take action to reverse the side-effect. What constitutes an inverse procedure depends upon the nature of the side-effect and the external object(s) affected. For example, if the `select_new` subtask draws a picture of the selected vehicle on the screen, an inverse procedure can be specified that erases the display if the subtask is later

undone. These techniques facilitate the interfacing with outside agents with the capability to handle, as best as possible, side-effects (Goal (5)).

Preserving and Re-using Information

One aspect of problem solving that our deduction method fails to deal with occurs when two alternative solutions share an identical subtask. In undoing one solution and trying the other, the work for solving the common subtask is lost If the `arrange_finance` subtask must be redone, due to a slight modification in the amount to be borrowed, a person should not have to once again enter his or her name and address.

One traditional approach to this problem in logic programming has been to use *assert* and *retract* predicates. A problem with this technique occurs when the clause that issued an assert is withdrawn due to backtracking. In our initial design of TICS we planned to include a special system predicate called a *soft-fact*. A soft-fact was to be a dynamically asserted fact that could be retracted. A soft-fact, unlike a dynamic fact generated by an evaluable predicate's procedure, could be used to solve more than one subtask. If a soft-fact was retracted, all subtasks that used information generated by that soft-fact would also be undone. Soft-facts could then be used to save and share answers generated by specified procedures.

A fact generated by an assert or soft-fact has an affect on the logic that depends upon when it temporally occurs during a deduction. Subtasks can use such dynamic facts only after they are generated and thus, the meaning of the program depends upon the temporal ordering of such events. We wished to avoid the

semantic problems associated with these predicates and so TICS supports other techniques, which can be implemented by the system designer, to share some information between subtasks. These answers can be retained by the system even if the subtask that generated the information is undone, (Goal (6)).

TICS' large-grain functionality results in deduction trees that are much smaller than traditional logical approaches. Small trees aid the designer in removing common information from multiple subtasks and placing this information into a single subtask that is as close or closer to the root of the deduction tree. The `person_ID` subtask illustrates this technique in the extended car-buying example presented in Chapter 5. In this example, the name and address of the buyer is required by both the `arrange_finance` and `do_tradein` subtasks, so a subtask called `get_buyer` is created to supply this factored-out information. If either `arrange_finance` or `do_tradein` are undone, the name and address information will persist.

TICS supports an additional technique, sometimes referred to as the *blackboard method*, to provide for persistent shared data. Standard logic predicates must use logic terms to compose and select information, but TICS' external processes can incorporate the functionality to handle formatted data, e.g., read and write records. The `arrange_finance` and `do_tradein` subtasks can contain a pointer to some common storage area, e.g., a file. These subtasks can then read and/or write the buyer's name and address to and from this storage area.

```
buy_car( ... ) :-
      do_tradein(file_name, ... ),
      select_new( ... ),
      arrange_finance(file_name, ... ),
      bottom_line_constraints( ... ).
```

Writing to a file is different than asserting a fact. A file's data is not necessarily available to all subtasks; file access can be restricted to only those subtasks with the file's name in their local environment. In addition, the choice of whether to use the file's data and how to interpret the data's meaning is left to each subtask's external process.

2.4. TICS meets Pascal

The mating of human and machine is a complex craft. Maintenance costs of many systems overshadow development costs. The user relationship to a system is said to account for about 60 percent of the maintenance problem and is critical to system success [Lientz and Swanson 81]. TICS attacks this issue by providing a flexible and forgiving interface that is user-oriented.

We describe how we used our prototype TICS system to improve the user-interface to a program written by Knuth to solve the following problem posed by Jon Bentley:

> The input consists of two integers M and N, with $M < N$. The output is a sorted list of M random numbers in the range 1..N in which no integer occurs more than once. For probability buffs, we desire a sorted selection without replacement in which each selection occurs equiprobably. [Bentley and Knuth 86]

Knuth wrote the code using his "literate programming" system called *WEB*. The language used by WEB is Pascal. This program is presented by Bentley as an

example of an elegant and efficient program.

The "short dialogue with the user," as Knuth calls it, comprises the largest single chunk of code in the program. The user-interface requires seventeen lines of code versus nine lines for the next largest routine Knuth's program first obtains from the user a positive integer "N", the size of the population. The program next gets a positive integer for "M", the size of the sample. The requirement that "M" is less than "N" and that "M" is less than or equal to some constant is then checked. Finally the program creates and prints the sorted list.

To show the utility of TICS we describe how it was used to improve what we feel is the weakest aspect of the program, the interaction between the user and the machine. The interaction in Knuth's program is constrained beyond the inherent nature of the problem due to the sequential nature of the programming language Pascal. Using TICS, the task is decomposed into separate subtasks to get the value of "N", the value of "M", check the constraints between "N" and "M", create the sorted list and print the sorted list. We express this decomposition with the following clause:

```
knuth(N, M, P) :-
      get_num('Pop. size, N =', N),
      get_num('Sample size, M = ', M),
      check(N, M, P),
      create_list(P, N, M, List),
      print_list(List).
```

The evaluable predicate `get_num` specifies a process whose purpose it is to ensure that the second argument is equal to a numeric value. This process first determines

the value of its second argument. If that argument is a numeric value, get_num does nothing additional. This scenario would occur if the value were supplied in the invoking goal as follows:

```
?- knuth(200, 100, P).
```

If get_num's second argument has not been constrained to a value, the process, using its first argument, prompts the user. After the user enters a value, get_num examines the value and if it is numeric, instantiates its second argument to that value. If the value was not numeric, then get_num informs the user of the error and re-solicits.

The evaluable predicate check specifies a process whose purpose it is to ensure that its first argument is greater than its second. The check process waits, if necessary, until its first and second arguments are bound to values and then checks the required constraint. If the constraint is satisfied, then check instantiates the variable P and terminates. If the second argument is greater than the first, check informs the user of the problem and asks whether the user wants to undo the value for N or for M to resolve the constraint violation. When the user specifies the variable he or she wants to change, check tells TICS to undo, via intelligent backtracking, the process that instantiated that variable. TICS will automatically undo check because check has read the value of a variable instantiated by the process being undone and is, therefore, causally dependent upon it. Since get_num and check are evaluable predicates of type external generator, TICS, after backtracking, will attempt to re-solve the initial procedure by re-invoking those processes.

The subtask `create_list` requests TICS to notify it when the user has input legal values, i.e., the variable `P` has become instantiated. The subtask waits until it receives notification and then establishes a file containing the sorted list of random numbers, writes the file's name to the variable `List` and terminates. When TICS notifies `print_list` of the instantiation of `List`, `print_list` will create a hard copy of the contents of the file named by the variable `List` and terminate.

We note the following advantages over Knuth's solution. Knuth's code required the value for `N` to be entered prior to the value for `M`. With TICS the input parameters are obtained via concurrently executing processes and could thus be specified simultaneously by the invoking procedure or in any order by the user. If the user entered a value for `M` that is greater than `N`, Knuth's program required that `M` be changed or the program aborted. With TICS, the subtask `check` allowed the user to change either `M` or `N`. Intelligent backtracking ensured that only the subtasks affected by the change were undone. Thus, if the user wishes to change the value of `N` the previously entered value of `M` remains valid.

TICS provided additional flexibility by allowing the task to be started with the value for some, all or none of its arguments supplied. The `get_num` subtask prompts the user only if its variable is uninstantiated.

2.5. System Development and TICS

Human limitations make it impossible to juggle too many things at once and make it worthwhile to package complex ideas into procedures that are hierarchically organized. TICS' Horn clause syntax provides a declarative way for a designer to

hierarchically modularize a complex system to allow different levels of abstraction and details, (Goal (7)). Logic is not ideal for handling numeric computation and manipulating persistent storage. Other forms of programming are more suitable. TICS augments logic programming's ability to search for solutions by extending logic predicates to include procedures implemented as independent processes. The internals of these processes and their parameters are hidden and isolated from the logic. These procedures can be implemented in any language and in any operating environment capable of interfacing to a TICS system. Thus, the designer is given the flexibility to develop a system using a wide variety of tools and environments, (Goal (8)). This flexibility in mixing languages and operating systems helps support the easy re-use of existing solutions, (Goal (9)). The ability to re-use modules from system to system can provide a standard set of easy-to-use tools that promotes a continuity of style between applications.

TICS uses logic to decompose problems into subtasks. This encourages and enhances the development of software for diverse groups of end-users by incorporating the ability to separate the user-interface processes from application processes, (Goal (10)). In addition, modularization also allows interface programmers to implement the interactive displays iteratively and in parallel with the development of the rest of the program, a goal espoused by Norman and Draper [Norman 83] [Draper and Norman 84].

The division of functionality into separate unit processes provides the capability to distribute the solving of subtasks to multiple processors, (Goal (11)). One use

of this ability would be to offload processing intensive interface-specific routines, such as low-level graphics, scrolling, text editors and menu routines, from the main processor onto workstations. TICS provides the ability for procedures to interact via powerful communication and control synchronization techniques. These techniques are described further in Chapter 4 and allow for the incorporation of stream communication, as described by Kieburtz and Nordstrom [Kieburtz and Nordstrom 85] and coroutining as in Kahn and MacQueen [Kahn and MacQueen 77].

Chapter 3

Related Research

To define and design our user-oriented problem solving system we examined and drew upon knowledge from many areas of related research. In this chapter, the task composition methods provided by office management and user-interface management systems (UIMS) and some techniques used to formally specify programs are discussed. Next we examine extending logic programming by incorporating other languages and pursue the issues of concurrency and backtracking. Finally, other systems are presented that provide general user-undo facilities.

An intelligent user-interface called POISE was designed to address the problems of task definition and support [Croft and Lefkowitz 84] [Broverman and Croft 85]. POISE can run both as a task planner, taking stated task goals and directing the user through sequences of actions designed to achieve those goals, or as an interpreter, to recognize user actions and assist where appropriate. TICS only proposes to do the former. POISE, like TICS, handles tasks that involve complex sequences of actions that do not correspond to running simple tools but involve problem-solving ability. A version of an Event Description Language is used to specify procedures. A specification contains a section describing constituent procedures and their sequencing, a section to describe task attributes in terms of semantic database items, constant values or attributes of its constituent procedures, a section on constraints, and finally, sections describing preconditions and satisfaction criteria. TICS' Horn clause

specification, we feel, captures all this information in a clearer and more direct and declarative manner.

POISE uses three major components, a planner, focuser and interpreter. These components provide a means to direct the user through a sequence of actions to achieve goals, maintain constraints and incorporate backtracking to correct errors. Some of the initial ideas for TICS came from trying to understand the POISE system by casting it into logic programming. We believe that TICS' use of resolution theorem proving provides a simpler and more straightforward way to provide this task management functionality.

Much work has been done in the area of User Interface Management Systems (UIMS) to develop a framework to coordinate the interaction between user-interface and application modules. This work is usually referred to as dialogue management. Various techniques have been used to specify and implement this aspect of a system. These specifications can be classified into two broad categories, declarative and non-declarative. Some declarative ways of defining the dialogue include BNF grammars [Reisner 81], special purpose languages [Prywes and Pnueli 83] and first-order logic [Roach and Nickson 83]. Non-declarative approaches have included versions of state transition diagrams and event languages. Examples of these techniques are in Wasserman [Wasserman 85] and Green [Green 85] respectively.

Jacob argues that non-declarative specifications, and specifically state transition diagrams, are the most suitable for describing interactive human-computer interfaces because they represent time sequences explicitly [Jacob 83]. This may be

true when the possible sequence of events is rather limited. Concurrency complicates the issue immensely. Cardelli and Pike developed a system that handles concurrency using a set of communicating finite state machines [Cardelli and Pike 85]. Their scheme provides for efficient handling of a limited number of interaction devices, such as mice, buttons and keyboards, but as they remark,

> ... in general the state space of a set of concurrent processes can explode. [Cardelli and Pike 85]

There is no reason for the problem solver to anticipate or have knowledge of all possible event orderings. This detailed information will only confuse the user. We feel that to understand a complex concurrent program it is better to simply know what must be done. The step-by-step details of all the possible ways of how to achieve a goal are better ignored, i.e., abstracted away.

Formal declarative specification techniques allow one to prove the characteristics of programs that are developed through semantics-preserving transformations of such specifications. As Kowalski states,

> Logic reconciles the requirement that the specification language be natural and easy to use with the advantage of its being machine-intelligible. [Kowalski 82]

The meaning of a TICS specification is thereby automatically preserved. There is no need for a separate verification step. Davis compares two formal specification techniques, Guttag and Horning's algebraic equations and Horn clause logic [Davis 82]. She feels that the major distinction between the two methods is the way in which questions about the specification can be handled. The algebraic equation approach requires questions to be submitted to an *expert* who reformulates an informal user

query into a formal question from which an answer can be derived from the axioms of the specification. While this approach may also be used with Horn clauses, they can also be executed directly, to answer questions by running the specification and observing the results. Roach and Nickson assert that logic not only provides the full power equivalent to a Turing machine but it does so in a manner that is more easily modified than algebraic or functional methods because its statements are less closely coupled [Roach and Nickson 83]. They feel rule-based languages, like logic, provide for rapid development and easy modification by adding or modifying individual rules. Their paper describes the successful specification and development of a complex system using Prolog to manage the dialogue between user and machine.

Wadge and Ashcroft state that logic languages do not handle "bread and butter" computation well [Wadge and Ashcroft 85]. These languages lack the ability to define their own functions, and even the arithmetic operations are not really respectable. They go on to state that perhaps in the future "inference" programming languages (such as Prolog) and dataflow languages (such as Lucid) will be used in cooperation.

There are many examples in the literature of integrating logic and other languages. LOGLISP is a system that uses a non-backtracking approach to implement logic with functional programming techniques [Robinson and Sibert 82]. QLOG takes the approach of implementing logic as an embedded sublanguage of the host language Lisp [Komorowski 82]. As a result of this embedding, QLOG can easily incorporate many of the major components of the Interlisp environment. This

embedded approach differs from the more conventional freestanding implementations, such as TICS, where the host and embedded languages have more distinct syntactic and semantic structures. At the University of Salford, a system that combines Lisp and Prolog along with the ability to execute FORTRAN77 was developed because

> ... there exist programs which are easy to write in a conventional programming language but which are tortuous and obscure in Prolog. [Bailey 85]

A commercial system, developed at Texas Instruments, uses a LISP-based interpreter of Prolog [Srivastava 86]. Their system provides for the arbitrary exchange of data and control between LISP and Prolog. Moffat and Gray combine Prolog logic search strategy with PS-Algol, a highly typed language [Gray, Moffat and Boulay 85] [Moffat and Gray 86]. The use of PS-Algol allows them to implement efficient procedures to manage complex objects that do not fit easily into Prolog's logical model. A language such as PS-Algol allows the manipulation of complex objects such as semantic nets and efficient list structures.

TICS differs from all the schemes above by providing for the non-deterministic composition of procedures written in any language. These procedures can execute in any operating environment that is capable of communicating with TICS. TICS provides full support for intelligent backtracking and concurrent execution of logic terms. These features allow TICS to provide a system that models problem solving in a way that is natural for people to use and that imposes the minimum of arbitrary temporal constraints.

The problem of coupling logic to a language with formal data types must be addressed. Furukawa, et al. [Furukawa, Nakajima and Yonezawa 83] and Mycroft and O'Keefe [Mycroft and O'Keefe 84] developed systems that require type declarations for all of their logic procedures. TICS uses the technique of Moffat, Gray and Boulay to integrate typed data from other languages with a type-free logic. This technique,

> ... is more suitable when we want to hide a representation but do not want the extra discipline of typing our program. It could be considered as a primitive which could be used to implement a Furukawa type scheme at a later date if desired. [Gray, Moffat and Boulay 85] [Moffat and Gray 86]

The representation of types is isolated from the logic by making typed data private, their internals hidden and not directly accessible by logic. Unification is used to pass these terms safely to language units that can directly manipulate and maintain the correctness of the typed data. These features are more fully described in the next chapter.

A number of logic systems exist that permit the reduction of several sub-tasks in parallel. This form of parallelism is usually referred to as *and-parallelism.* Two such languages are PARLOG [Clark and Gregory 85] and a successor language, Concurrent Prolog [Shapiro and Takeuchi 83]. Both languages assume that the program makes the "correct" non-deterministic choices: once a goal has reduced itself using some clause, it is committed to that clause. This computational behavior is known as *committed-choice non-determinism.* Once a clause choice is committed to, by satisfying its guard conditions, it is never rescinded via backtracking. Modes are used in PARLOG and read-only and commit operators in Concurrent Prolog to

impose a direction on communication and provide procedure synchronization. A different approach to parallelism is taken by Conery and Kibler [Conery and Kibler 81] [Conery and Kibler 85]. They avoid the use of non-logical annotations, including guard conditions, but provide only limited and-parallelism. Borgwardt modifies their concepts to include special mode declarations to support efficient stream communication [Borgwardt 84].

All of these systems implement the small-grain functionality of standard logic predicates. TICS powerful external procedures incorporate large-grain functionality that can produce a simpler and more human-oriented task decomposition. TICS keeps the logic pure and simple by not describing data flow direction or temporal constraints at the specification level. Evaluable predicates can impose these constraints, when necessary in order to synchronize and communicate, by using the facilities of TICS' special purpose database.

The emphasis in TICS is on flexibility of interaction, not just efficiency of operation. This flexibility requires the inclusion of real non-determinism to provide alternative solutions, and backtracking to provide a means to undo user and system actions. To improve user-machine interaction, the fully intelligent backtracking system of *plan-based deduction*, [Cox and Pietrzykowski 81] [Forsythe and Matwin 84] [Matwin and Pietrzykowski 85], is used rather than the potentially more run-time efficient but less accurate semi-intelligent strategy of Chang and Despain [Chang and Despain 85]. A fully intelligent backtracking scheme allows TICS to provide the user with exact information about any resolution failure in addition to minimizing the

effects of backtracking. The large-grain functionality of TICS' external procedures provides for a sufficient level of efficiency despite the overhead imposed by these features.

Other systems provide different models to allow the user to cancel the effects of past actions and to restore an object to a prior state. COPE is an editing and incremental program development system that provides for user recovery and reversal [Archer, Conway and Schneider 84]. INTERLISP also provides for extensive recovery capabilities using a similar but slightly different model [Teitelman 75].

The COPE model describes the basic interaction cycle as having two logical phases. In the first phase the user edits a script. A script represents a modifiable specification of a transformation. Submitting the edits begins the second phase where the system performs some execution, e.g., undoing and rebuilding tasks, in order to ensure that the state of the system will correspond to that specified in the script. This cycle is repeated until the user deems the task complete. COPE's recovery mechanism is based upon a modified truncate and reappend version of their general model. Basically, recovery involves undoing all system changes required to return the system back to a state that was valid prior to the time of the operation that is being undone. The commands occurring after that point in the script are then executed. Since these commands were modified during the first phase the resulting state corresponds to the new script specification.

The task of incremental program development with COPE is composed of the alternate invocation of user-edit and program-execution subtasks. This composition

causes each subtask to be dependent upon those subtasks solved before them. The temporal order of solutions imposes a linear chain of causality. COPE's recovery scheme, similar to the linear backtracking of standard Prolog, is well suited for this problem domain. TICS' domain of general problem solving requires the composition of subtasks with more complicated inter-relationships. The intelligence of TICS' dependency-directed backtracking scheme is needed to avoid the waste and user confusion of blind linear backtracking.

TICS is based on a database with built-in functionality that controls and coordinates the composition of individual procedures and the flow of data between them. To provide communication and synchronization TICS uses techniques similar to those found in Humanizer [Grossman 85] [Maier, Nordquist and Grossman 86].

Chapter 4

What Makes TICS Tick

The key to TICS' implementation is its database, which incorporates special-purpose functionality and is main-memory resident. The database contains the system's specification, provides dynamic working storage and implements a *deduction engine* with the functionality of extended plan-based deduction. Everything in TICS, except the external processes for evaluable predicates, is contained within the database process.

Subtasks specified by evaluable predicates utilize external procedures that are implemented as separate processes. These procedures are used to solve subtasks and communicate with the database via messages handled by the Database Access Manager (DAM). Figure 4-1 is an overview of TICS in an operating system environment that supports multi-tasking and interprocess communication.

4.1. Deduction Engine

The deduction engine is based upon a method of resolution theorem proving called *plan-based deduction.*

> Attempts to find a refutation(s) are recorded in the form of plans, corresponding to portions of an AND/OR graph search space and representing a purely deductive structure of derivation ... It is proven that the algorithm is complete in the following sense: if for a given base a resolution refutation exists, then this refutation is found by the algorithm. [Matwin and Pietrzykowski 85]

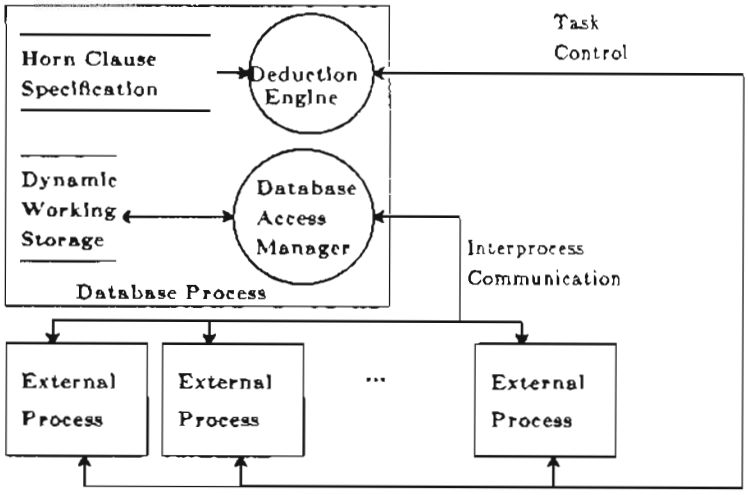


Figure 4-1. TICS Overview

The following is a simplified summary of how plan-based deduction works. The data structures used to implement this technique are described in more detail in Chapter 6. Plan-based deduction explicitly represents the current state of a goal's resolution as a *plan*. A plan consists of a *deduction graph* and its *potential function* and associated *constraint graph(s)*. The deduction graph provides information about which clause was used to solve which goal. The potential function assigns to each goal in the associated deduction graph the subset of the program's Horn clauses, i.e., base, that may be used in the future to solve that goal. This subset consists of those clauses whose heads unify with the goal and that have not been used in the past with that goal. Such a representation allows for a flexible search strategy. Unsolved goals can be solved in any order. In contrast, many logic programming implementations limit the search strategy to depth-first-search. This limitation provides

implicit information that allows them to use stack-based implementations of refutation procedures.

In plan-based deduction, a constraint graph represents the unification constraints imposed by the various resolution steps on a logic variable. A set of constraint graphs is associated with each deduction graph; each constraint graph describes one group of logical terms, i.e., variables, constants or structures, bound together by unifications. This method is less space efficient than Prolog's use of a binding array and trail. However, Prolog's method does not retain the information to record which unification of variables caused a particular binding, nor can the binding be undone except in a "linear" manner. Plan-based deduction's constraint graph provides the information to identify the actual source or sources of a variable's binding. Knowing the sources gives plan-based deduction the ability to determine data unification dependencies and thereby implement intelligent backtracking. In addition, using plan-based deduction a goal can be solved with a clause that, due to bindings established earlier, will cause a unification failure. By allowing this unification, plan-based deduction, unlike Prolog, can force the earlier binding to be undone.

In plan-based deduction an initial *current* plan is generated containing only a single root node. This node consists of the original query, i.e., initial goal(s), and its potential. As long as there are no detected unification conflicts in a current plan's associated constraint graphs and there are remaining unsolved goals, the current plan is developed. Development proceeds by adding nodes to the deduction graph

that represent the resolution of an unsolved goal with a clause from its potential. When all goals are solved a solution is achieved. When a unification conflict is detected, plan-based deduction determines the **set** of sets of clauses that can be undone to remove the conflict and still provide a non-empty potential for all unsolved goals. The elements of the **set** represent the alternative sets of clauses to undo during backtracking. Undoing any set of clauses causes the *pruning* of the corresponding arcs and nodes of the current plan's deduction graph, and thereby results in the creation of a new plan. New plans are inserted into a *store* for future use. A new plan can be created for each element of the **set** of sets of clauses. The current plan is then discarded and a new current plan is selected from the store. If additional solutions to the original goal are desired, *artificial conflicts* are added to force solved goals to be re-solved, thus generating any additional solutions. The algorithm can continue until all plans are solved or annihilated.

TICS implements the following processing strategies in its version of plan-based deduction. A check for unification conflicts is made after every goal resolution. The selection of which unsolved goal is to be developed next is either made automatically by the system, on the basis of assigned priorities, or chosen from a menu by the user. This mechanism is described more fully in the following chapters. TICS' resolution algorithm is oriented to finding single solutions, but retains the capabilities to find alternative solutions. Plan-based deduction is simplified by TICS' restriction of the base to allow only Horn clauses.

TICS extends plan-based deduction by allowing for and-parallel resolution. All unsolved goals can be solved concurrently. TICS provides procedures executing in parallel with the ability to dynamically examine the evolving environments of other procedures with which they share common variables. This sharing provides procedures with the ability to use the maximum amount of information available in order to limit their range of possible solutions. However, the accessing of a value by a procedure makes that procedure causally dependent upon that value. If the procedure that established the value later fails and there are no other procedures that have also established that value, then the procedure that accessed the value must be undone. The establisher has, in effect, *caused* a change in the accesser's environment. TICS' database must track these causality dependencies. For example, if procedure `p1(A, B)` has accessed the value of `A` that was instantiated by procedure `p2(A)` and by no other procedure, then if we fail `p2`, we must also fail `p1`. However, if `p1` has not gotten around to reading the value of `A`, then there is no reason to fail it. We want `p1` to be able to access the most current unified value of `A` because it may be able to use that knowledge to generate a value for `B` that is more reasonable then a value generated without that knowledge.

4.2. External Procedures

A predicate can be specified as being an evaluable predicate via the `$pred` system predicate, described later. TICS attempts to solve a goal against an evaluable predicate by executing the predicate's external procedure. This procedure is implemented as a separate process. An external process that completes and

terminates with success represents an assertion , i.e., a generated fact, that solves the goal against its evaluable predicate. A process that terminates with failure means that with the current variable bindings, nothing could be asserted. If an external procedure terminates with failure TICS invokes backtracking to remove the goal.

External processes terminate upon completion. TICS also automatically terminates external processes if the evaluable predicate's node, i.e. clause, is removed during backtracking. A node will be removed if any of the node's goals, i.e. subtasks, cannot be solved. An evaluable predicate can be associated with a special type of external procedure called an *external generator*. An external generator represents a lazy generator of assertions. External generator processes that complete with success can suspend themselves rather than terminate, and thus have a memory of past actions. After backtracking, TICS can send a message to the external generator requesting the process to unsuspend and generate another fact. Thus, the process acts as a generator of facts on demand. An external generator can indicate to TICS that it cannot generate the requested fact by completing with a failure status. TICS will then invoke backtracking to remove the external generator's goal from the deduction.

There is a deliberate similarity between the way an external generator procedure and a human operator provide data for a program. When a user supplies input, he or she, in essence, asserts a fact. If, in the future, this input is unsatisfactory, the user is again asked to enter a new value. The user can be viewed as a lazy

fact generator whose cranial memory stores local state information regarding past actions. It should be noted that there is nothing in TICS that prevents an external generator from generating the same fact more than once.

4.3. Data Access Manager (DAM)

TICS' database does not provide persistent storage but is a database in the sense that it provides controlled sharing of data. All access by external procedures to this data is via the Data Access Manager (DAM). TICS' logical terms are read and written by external procedures via commands sent to and received from the DAM in a manner similar to that used in the Humanizer framework [Grossman 85] [Maier, Nordquist and Grossman 86]. External procedures are invoked with the database identifiers of the logical terms they can access. These terms are accessed only via database read and write messages.

When a procedure issues a read command for a term in its environment, the DAM responds with the term's value. Since plan-based deduction does not store variable bindings with destructive updates, the current value of a variable is derived by unifying the variable's constraint graph. If the value is unacceptable to the procedure, then the procedure can either try some alternative action or terminate with a failure status.

When an procedure sends a write command to a variable, the database's DAM enters that value into the constraint graph of the appropriate variable. If the database cannot unify all terms in the constraint graph, intelligent backtracking is invoked. Unifiability is restored by selecting a set of nodes in the current plan's

deduction graph to be undone.

Each external procedure is responsible for acquiring any data it needs. When a procedure requires data from one or more logical variables that are not yet instantiated, the procedure can issue requests to be notified when those variables change and then suspend itself. This mechanism permits external procedures to be started even if the data they require is not currently available. *Mode* declarations and other annotations to provide data synchronization are neither permitted nor required in the logic. Sometimes logical variables will never be fully instantiated nor even accessible, as in the case of infinite structures and non-terminating computations. Such cases cause no problem as long as the missing information is not required by a procedure.

4.4. Data Types

Type checking of an external procedure's parameters, i.e., database identifiers of the logical terms the procedure can access, is the responsibility of the individual procedure. Data types can be protected from TICS' type-free logic by being specified as being private terms. This technique is used in Persistent Prolog [Gray, Moffat and Boulay 85] [Moffat and Gray 86]. Private terms cannot be examined by TICS' logic because such a term can only be unified with another term of that type or with a free variable. TICS' logic can only be used to pass private terms from one procedure to another or to construct compound terms, e.g., lists, made up of private terms. Special data objects can be represented by private terms. We can create a *handle* to an object by binding a private variable to a pointer to that object. Only

external procedures that know a special object's handle, i.e., have a variable bound to the object's term in their environment, can access it.

Coercion of data can be done internally by each procedure or through the use of special predicates. For example, to transfer data between two different procedures, p1 and p2, each of which requires different data types, a procedure called `coerce_it` can be written and used as follows:

```
p1(AIn, AOut), coerce_it(AOut, BIn), p2(BIn, BOut)
```

The `coerce_it` procedure can be written so as to be bidirectional.

4.5. Managing Freedom

In the introduction, we argued the need for freedom in problem solving. Systems that impose highly structured interaction limit a person's freedom to use his or her individuality, creativity and originality to solve problems. TICS was conceived as an interactive problem-solving system with minimal constraints. However, certain human-oriented constraints on problem-solving were used to simplify parts of its design. People normally use one approach at a time to solve a problem. Therefore, TICS is *or-sequential*, a goal is solved by only one clause at a time. Horn clause logic, rather than full first order logic, was chosen because it is more efficient to implement and provides a convenient way for people to naturally express and understand the decomposition of problems. Despite these restrictions, TICS provides a lot of freedom, much more than traditional approaches, as was demonstrated by the example of Knuth's Pascal code.

While we still believe in freeing the user from artificial constraints, even simple Horn clause specifications, such as our *car buying* example, specify a very large space of possible solutions. Our initial implementations made us realize that a person can easily be overwhelmed and burdened by too many choices presented all at once An exhaustive search of such a space is infeasible.

TICS' original design was to have the end-user provide the meta-level reasoning to guide the deduction. The end-user's general and domain-specific knowledge was to direct the search for a problem's solution. After TICS was implemented and the first examples run, it became apparent that too much freedom can be confusing and annoying. To solve a problem, many questions arise. To ease the burden on the end-user, we enhanced TICS to provide the system designer the option to flexibly limit the number and structure the presentation, of those decisions that an end-user must make. A TICS system designer can specify that some of these decisions be made automatically by the system. In addition, the designer can provide information to guide both the system and end-user in choosing among alternatives.

During forward deduction operation, TICS' deduction engine, solves subtasks. Whenever a unification conflict is detected, the deduction engine invokes the backtracking operation to undo subtask(s) to remove the conflict. Forward deduction is then resumed. These operations are described in further detail in Chapter 6. The deduction engine can be set to operate in either a global automatic or global manual mode of operation during forward deduction and backtracking by the `$deduction` and `$backtrack` system predicates, respectively. If the system is set to the global

manual mode of operation for forward deduction the choice of subtask to solve next and the method to use to solve it are always made by the user. In terms of logic, these questions refer to the choice of subgoal to solve and the clause to use to solve it. This feature is useful for single-step operation during system debug by the designer but usually asks too many unimportant and meaningless questions to be useful for a user during a typical problem-solving session.

If the system is set to global automatic mode of operation, the system makes all selections. The system designer can assign forward priority values to clauses and predicates to direct TICS' selections. This all-or-nothing form of control was found to be too inflexible. Therefore, we provided a mechanism to permit the designer to specify that certain predicates, even in automatic mode, are to be solved by asking the user to select the clause to use. This specification is done using the `or_mode` field of the `$pred` system predicate.

In backtracking, the system may find multiple ways to restore unifiability; each alternative generates a new plan that could be further developed. If automatic mode is set for backtracking the choice of which plan to develop next is made by the system on the basis of backtracking priority. If manual backtrack mode is set, the user is asked to make the choice.

TICS incorporates and-parallelism. Therefore, many subtasks can be executing concurrently. Sequentialization and synchronization can be accommodated by methods previously described. Computations may be put into a single external procedure that can use local state to order events. Alternatively, evaluable predicates

can use logical variables to coordinate their activities. These techniques alone do not provide the designer with the means to flexibly adapt the system to available resources. For example, the number of simultaneously executing tasks that interact with a human or CRT may need to be limited to prevent exceeding the user's mindspace or the CRT's screen space. To remedy this problem, we enhanced TICS to provide the designer with a *gate* mechanism to control the number of tasks that access a particular resource at any instant.

To enhance communication between the system and the user, TICS was provided with a menu facility. Menus are dynamically generated by the system when the user has to be queried about which way the deduction should proceed. Menu choices are listed in priority order. Most people do not understand the syntax of logic and so we have provided the designer with the option of providing a natural language format for each predicate and clause. TICS maintains the information required to answer the questions, "How did we get to the current state?", and "Why are you asking this question?". Since plan-based deduction maintains the current deduction tree, there is no need to pass the context of the deduction as an extra argument, as would be required in Prolog [Clark and McCabe 82] [Walker 82]. At this time we have not yet fully implemented this explanation facility. Gates, clause selection and natural language formats are illustrated and described further in the next chapter.

Chapter 5

Examples

The following programs were designed to demonstrate the power of TICS' logic to compose interactive tasks to solve problems and the ability of TICS' intelligent backtracking to allow system- and user-initiated changes. We use these examples to introduce and describe TICS' system predicates. The Appendix contains a summary of these predicates.

5.1. Car Buying

The *Car Buying* problem consists of trading in a used car, selecting a new car and financing the balance due. This problem is basically the same one we discussed in Chapters 1 and 2 with some details added. For simplicity, we initially show only the basic Horn clauses to illustrate how the problem is decomposed into subtasks. We will then introduce the complete program.

To buy a car: get buyer information, establish trade-in's worth, select the new vehicle and arrange financing. Ensure that cost of new vehicle less trade-in does not exceed the maximum amount to be financed:

```
buy_car(
  person_ID(Name, Income),
  tradein(TYear, TVehicle, TMethod, TWorth),
  new_car(NYear, NStyle, NColor, NPrice, NDelivery),
  finance(FMethod, FMax, FAmount, MonthlyPayments))  :-
    get_buyer(Name, Income),
    do_tradein(Name, TYear, TVehicle, TMethod, TWorth),
    select_new(NYear, NStyle, NColor, NPrice, NDelivery),
    arrange_finance(Name, Income, FMethod, FMax, FAmount,
      MonthlyPayments),
    bottom_line_constraint(TWorth, NPrice, FMax, FAmount).
```

Trade-in can be either handled by the dealer for wholesale book value:

```
do_tradein(Name, Year, Vehicle, dealer, Worth)  :-
    book_value(low, Year, Vehicle, Worth).
```

... or by private sale for retail book value:

```
do_tradein(Name, Year, Vehicle, private, Worth)  :-
    book_value(high, Year, Vehicle, Worth).
```

The system predicate `is` is used to set `Year` to 1987:

```
select_new(Year, Style, Color, Price, Delivery)  :-
    is(Year, 1987),
    find_vehicle(Style, Color, Price, Delivery).
```

Determine the new vehicle by either choosing from stock:

```
find_vehicle(Style, Color, Price, 0)  :-
    in_stock(Style, Color, Price).
```

... or by the user placing an order for a specific style and color:

```
find_vehicle(Style, Color, Price, DeliveryTime)  :-
    get_style(Style, Color),
    get_color(Style, Color),
    order_vehicle(Style, Color, Price),
    order_time(DeliveryTime).
```

Use the following fact to order the vehicle via the delayed side-effect predicate described later in this section:

```
order_vehicle(Style, Color, 15000).
```

The following facts denote the cars currently in stock:

```
in_stock(coupe, white, 12000).
in_stock(sedan, black, 11000).
```

For simplicity we assert that all ordered vehicles have the same delivery time. The following fact specifies the delivery time for an ordered vehicle:

```
order_time(90).
```

Note: `get_buyer`, `book_value`, `get_style`, `get_color`, `arrange_finance` and `bottom_line_constraint` are evaluable predicates solved by external generator processes, described later in this section.

The rest of this section will detail the complete *Car Buying* program interspersed with an explanation of the system predicates used to manage freedom. TICS can accept unannotated Horn clauses. However, control and descriptive information about individual predicates can be specified optionally by including `$pred` assertions of the form:

```
$pred(name(variable1, variable2 ...), [Field1, Field2 ... ])
```

The first argument of `$pred` is a structure that identifies the predicate being annotated. The functor, `name`, is an atom whose value is the name of the annotated predicate. The subterms of the `name` structure are variables; the number of these variables is equal to the arity of the annotated predicate. The second argument contains a list of one or more optional `Field` terms in any order. Each `Field` specifies one attribute. Certain attributes are incompatible with others, as described below. An attribute may associate a priority with a predicate or clause. In TICS, priorities are positive integers, the higher the number the higher the priority.

To prevent system resources from being overwhelmed, TICS provides the system designer with the option of specifying *gates*. Gates are used to limit the number of goals that can access the user or other resource at any instant. These gates are declared by the `$gate` system predicate whose arguments specify the resource's name, the maximum number of goals that can access the resource at any one time

and the manner in which the goals are to be selected from the gate. Goals can either be selected by the user from a menu or selected automatically by the system, based upon the gated goals' priorities. The `$pred's gate` attribute specifies that a predicate's goals use a limited resource and assigns those goals a priority of access to that resource. In addition, predicates with the `$pred's or_mode` attribute have their goals automatically assigned to the `select` gate. When the system attempts to solve a subtask that is gated. the system initially inserts that subtask into the appropriate gate according to the task's gate priority. After all goals have been initially serviced, i.e., solved or inserted into gates, the system will remove and solve gated goals, one at a time. Gates are examined in the order they are initially declared. The first gate found whose limit is greater than the number of subtasks currently accessing the gate's resource is selected. If the gate's selection mode was specified as `manual`, the user chooses a goal from a menu containing all the gate's members. If the gate's mode is `automatic`, the system will select the highest priority goal.

The order of initial gate declaration is especially important when the `select` gate is used. `Select` goal's may establish additional, possibly gated, goals. Gates declared after the `select` gate will not start solving their goals until all of the gate's goals, including those established by `or_mode` predicates, have been inserted. Our first two examples declare the gates in different orders to accomplish different purposes. The *Car Buying* program will declare the `select` gate first because the procedure used to solve `find_vehicle`, an `or_mode` predicate, may establish the

goals, get_color and get_style. This gate ordering ensures that these goals will be inserted, in priority order, into the crt's gate before the crt's gate is initially serviced. Thus, get_color and get_style will be invoked before lower priority goals, e.g., bottom_line_constraint. This behavior is desired because the lower priority goals require information that is based upon the data supplied by get_color and get_style.

The following $gate predicate establishes the select gate. This predicate states that the system will automatically choose the next or_mode predicate's goal to solve. The select gate limit is always one:

```
$gate(select, 1, automatic).
```

Limit to three the number of subtasks using the display at any one time. The three are automatically selected by the system on the basis of each predicate's $pred gate priority:

```
$gate(crt, 3, automatic).
```

The following $pred system predicate contains the clauses and pdesc annotations for the buy_car predicate. The clauses term specifies the procedures that can be used to solve the buy_car predicate. Each procedure is described by a clause term.

Forward deduction priority is the second argument of the clause term and is used in determining the procedure to solve a goal. If clause selection is manual because either the deduction mode is set to manual or the predicate has an or_mode manual annotation, then the user is presented with a menu with the goal's potential clauses listed in priority order. If clause selection is automatic

because both the deduction mode is set to automatic and the predicate does not have an or_mode manual annotation, then the system automatically selects the highest priority procedure

Backward deduction priority is the third argument of the clause term and is used to determine which predicate to undo during backtracking. If there is more than one element in the **set** of sets of goals that can be undone to restore unifiability, then the sets are ordered by the sum of the backtracking priorities of each set's members. If backtracking mode has been set to automatic, then the system undoes the set with the lowest sum of backtracking priorities. If backtracking mode has been set to manual the user selects the set from a priority-ordered menu. The fourth argument of the clause term contains the clause's natural language format. The fifth argument specifies the clause's information level that will be used by the, not as yet implemented, explanation facility. The information level will be used to control the amount of detail provided the user. The pdesc term specifies the natural language description of the predicate and has the same format as the last two arguments of the clause term.

The buy_car subtask is solved by a clause, i.e., procedure, described to the user by the system as "tradein, select and finance a new car". The predicate buy_car is described by the phrase "buy a car: <1>, <2>, <3>, <4>"; where the current bindings of the variables Person, Trade, New and Finance are displayed in positions <1>, <2>, <3> and <4>, respectively. If the variable is unbound then "<unbound>" will be displayed in that position. For example, if Person were bound to *Jay* and the other three variables were unbound, then the following would be printed:

```
buy a car: 'Jay', <unbound>, <unbound>, <unbound>
```

When the display level feature is implemented, this description will only be printed if the current display level is less than or equal to 2:

```
$pred(buy_car(Person, Trade, New, Finance), [
    clauses([
     clause((buy_car(
       person_ID(Name, Income),
       tradein(TYear, TVehicle, TMethod, TWorth),
       new_car(NYear, NStyle, NColor, NPrice, NDelivery),
       finance(FMethod, FMax, FAmount, MonthlyPayments)) :-
         get_buyer(Name, Income),
         do_tradein(Name, TYear, TVehicle, TMethod, TWorth),
         select_new(NYear, NStyle, NColor, NPrice, NDelivery),
         arrange_finance(Name, Income, FMethod, FMax, FAmount,
           MonthlyPayments),
         bottom_line_constraint(TWorth, NPrice, FMax, FAmount)),
       1,1,['tradein, select and finance a new car'],2)
     ]),
     pdesc(['buy a car: ', Person, '  ', Trade, '  ', New,
       '  ', Finance], 2)
]).
```

The ptype term for get_buyer specifies that get_buyer is an evaluable predicate of type external generator and that it should be solved by executing the process named gb. The backtrack priority of this predicate is 6. The gate term says that get_buyer utilizes the crt and has a priority of access to this gated resource of 6:

```
$pred(get_buyer(Name, Income), [
    ptype(external_generator, gb, 6),
    pdesc(['buyer is ', Name, ' with income ', Income], 2),
    gate(crt, 6)
]).
```

The clauses term for do_tradein specifies two procedures that can be used to solve it. Note that not only do the descriptions of these clauses differ, but the first clause instantiates the variable Method to dealer while the second instantiates this variable to private:

```
$pred(do_tradein(N, Y, V, Method, W), [
    clauses([
     clause((do_tradein(N, Y, V, dealer, W) :-
        book_value(low, Y, V, W)), 2, 1, ['use dealer tradein'], 2),
     clause((do_tradein(N, Y, V, private, W) :-
        book_value(high, Y, V, W)), 1, 1, ['use private sale'], 2)
    ]),
```

The or_mode term states that when the system encounters a do_tradein goal to insert it into the select gate with a priority of 2. When a do_tradein goal is removed from the gate, because the or_mode's first argument is automatic, the system will choose the procedure to solve the goal from that goal's potential. The system will select the procedure with the highest forward priority. For do_tradein, the procedure in the first clause term has a higher forward priority than the procedure in the second clause term:

```
  or_mode(automatic, 2),
    pdesc(['determine trade-in value for vehicle ', Y, V], 2)
]).


$pred(book_value(Method, Year, Vehicle, Value), [
    ptype(external_generator, bv, 3),
    gate(crt, 3),
    pdesc(['old car is worth ', Value, ' using ', Method,
      ' trade-in '], 2)
]).


$pred(select_new(Y, S, C, P, D), [
    clauses([
     clause((select_new(Y, S, C, P, D) :-
        Y is 1987, find_vehicle(S, C, P, D)), 2, 2,
        ['select new car'], 2)
    ])
]).


$pred(get_style(Style, Color), [
    ptype(external_generator, 'gs', 3),
    pdesc(['style of car is ', Style], 2),
    gate(crt, 4)
]).
```

```
$pred(get_color(Style, Color), [
    ptype(external_generator, 'gc', 3),
    pdesc(['color of car is ', Color], 2),
    gate(crt, 4)
]).


$pred(find_vehicle(S, C, P, D), [
    or_mode(automatic, 1),
    clauses([
     clause((find_vehicle(S, C, P, 0) :-
        in_stock(S, C, P)), 1, 2,
        [' new car from stock, price ', P], 2),
     clause((find_vehicle(S, C, P, D) :-
        get_style(S, C), get_color(S, C),
        order_vehicle(S, C, P), order_time(D)), 2,
        2, ['ordering new car, price ', P, ' delivery ', D], 2)
    ])
]).
```

The `side_effect` term specifies that, when the user accepts a `buy_car` solution, the `order_vehicle` subtask in the final deduction tree should have the process named `o_letter` executed with the current bindings for that `order_vehicle`'s `S`, `C` and `Price` variables passed as arguments. The `side_effect` annotation is used in our example to send an order letter with the values for the selected vehicle's style, color and price:

```
$pred(order_vehicle(S, C, Price), [
    clauses([
     clause((order_vehicle(S, C, 15000)), 1, 8,
       ['order vehicle'], 1)
    ]),
    pdesc(['order vehicle ', S, ' color ', C, ' price ',
      Price], 1),
    side_effect(delayed, o_letter, [S, C, Price])
]).
```

The `or_mode` term causes TICS to insert `in_stock` goals into the `select` gate with priority 2. When an `in_stock` goal is removed from the gate, the user will select the procedure to solve the goal from a menu. The menu entries will consist of the goal's potential displayed in natural language format and in forward priority order:

```
$pred(in_stock(S, C, Price), [
    clauses([
      clause((in_stock(coupe, white, 12000)), 1, 4,
        ['white coupe 12000'], 1),
      clause((in_stock(sedan, black, 11000)), 2, 4,
        ['black sedan 11000'], 1)
    ]),
    or_mode(manual, 2),
    pdesc(['in stock vehicle ', S, ' color ', C, ' price ',
      Price], 1),
    side_effect(delayed, o_letter, [S, C, Price])
]).


$pred(arrange_finance(Name, Income, FMethod, FMax, FAmount,
      MonthlyPayments), [
    ptype(external_generator, 'af', 2),
    pdesc([FMethod, 'potential financing is ', FMax,
      ', actual financing is ', FAmount, ' with payments of ',
      MonthlyPayments], 2),
    gate(crt, 2)
]).


$pred(bottom_line_constraint(TWorth, NPrice, FMax, FAmount), [
    ptype(external_generator, 'blc', 9),
    pdesc(['finance amount ', FAmount, ' (new car price: ',
      NPrice, ' less trade-in: ', TWorth,') must be less than ',
      FMax], 1),
    gate(crt, 2)
]).


order_time(90).
```

Forward deduction in automatic mode will allow the system to select the goals to solve and the procedure to solve a goal unless unless otherwise specified by a `$pred` `or_mode` or `gate` annotation. The following clause is used to set global forward deduction mode to automatic:

```
$deduction(automatic).
```

Setting global backtracking to manual will allow the user to select the set of goals to be undone when unification constraint violations require backtracking:

```
$backtrack(manual).
```

This example program is illustrated by the following problem-solving scenario. The user enters the initial goal:

```
?- buy_car(
  person_ID(Name, Income),
  tradein(TYear, olds, TMethod, TWorth),
  new_car(NYear, NStyle, NColor, NPrice, NDelivery),
  finance(FMethod, FMax, FAmount, MonthlyPayments)).
```

The buy_car procedure that solves this goal creates the following goals: get_buyer, do_tradein, select_new, arrange_finance, and bottom_line_constraint. The get_buyer subtask is inserted into the crt gate. The do_tradein, an or_mode annotated subtask, is inserted into the select gate. The select_new procedure establishes 1987 as the year of the new vehicle and generates a new goal, find_vehicle, which is inserted into the select gate. Both arrange_finance and bottom_line_constraint subtasks are then inserted into the crt gate. This situation is illustrated in Figure 5-1A.

The select gate was specified before the crt gate, thus, the do_tradein goal is serviced first, causing a new subtask, book_value, to be inserted into the crt gate. The final select gate goal, find_vehicle, is then solved, resulting in the goals get_style and get_color being put into the crt gate. This situation is illustrated by Figure 5-1B.

TICS, as per our $gate specification, allows only three subtasks to access the crt at once. In this case we had specified that the system is to make the selection based upon assigned forward priority. Therefore, the goals get_buyer,

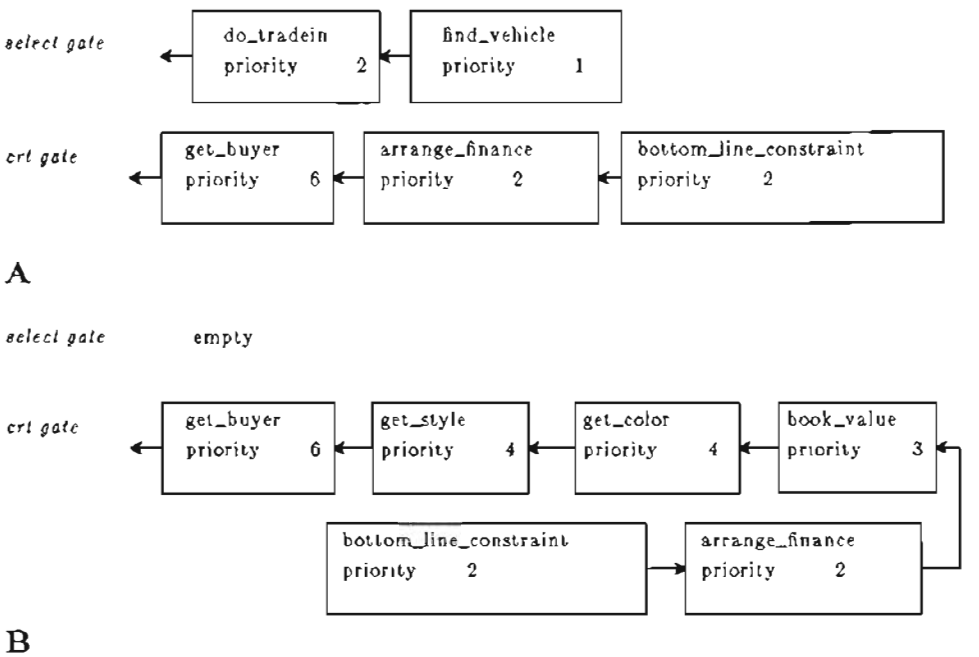


Figure 5-1. *Car Buying's* Gates

get_style and get_color with priorities 6, 4 and 4 respectively, are removed from the crt gate and solved by their external processes. These processes run concurrently and interact with the user via separate windows on the crt. The user can solve these subtasks in any order. In our scenario, the user chooses "black" as the color of the new car. The get_style subtask uses this color information to modify its prompt to only those styles available in black, i.e., coupe and sedan. This interaction illustrates that the two processes, get_style and get_color, cooperate by sharing Color and Style information.

The completion of the get_color subtask reduces the number of subtasks accessing the crt to below the gate's limit. Therefore, TICS removes book_value from the crt's gate and solves the goal by executing book_value's associated external process. This process needs only to question the user regarding the trade-in's year because the initial buy_car goal instantiated the trade-in vehicle's type information to "olds". The user then enters "coupe" for the new vehicles body style and continues to interact with the external processes until the remaining goals are solved.

Had the first find_vehicle clause been used to solve select_new, the subtask in_stock would have been generated rather than get_color and get_style. Because in_stock is specified as being or_mode(manual, 2) the user would have been allowed to select the clause to solve this subtask from a menu, i.e., choose the in stock vehicle. The entries in the selection menu would have been ordered by clause priority and displayed in the specified natural language format as follows:

```
1-  white coupe 12000
2-  black sedan 11000
```

The maximum amount that the buyer can finance is determined from the buyer's income by arrange_financing. The final net cost, i.e., price of new car less trade-in, must not be greater than the amount the buyer can finance. This requirement is checked by bottom_line_constraint when all of its variables become instantiated. If the solution is illegal, bottom_line_constraint will

permit the user to selectively change the trade-in, finance or new car selection decisions via intelligent backtracking.

In our scenario an initial solution is produced. The user then decides to change the method of financing. She specifies that additional solutions are desired and thus, TICS invokes backtracking. Since `manual` backtracking mode was selected, the user selects the `arrange_financing` subtask from the backtracking menu choices. Had `automatic` mode for backtracking been set, the system would have made the choice on the basis of backtracking priority. An advantage of intelligent backtracking is illustrated by noting that when a subtask is undone only those subtasks that depend upon it are undone. When `arrange_financing` is undone, `get_buyer`, `do_tradein` and `select_new` are unaffected. However, `bottom_line_constraint` is undone because it read the value of `EMax` established by `arrange_financing`.

The user enters the new financing information and `arrange_financing` generates a new value for `EMax`. The rerun `bottom_line_constraint` subtask examines `EMax` and determines that the new vehicle can be purchased. The user then accepts the solution and the `side_effect` process associated with the `order_vehicle` subtask is executed with the specified variables' current bindings. This process sends mail ordering the specified vehicle, a black coupe.

5.2. Plumber

The task of *Plumber* is to generate a report from one or more sources of data. To illustrate TICS' ability to incorporate existing functionality, our goal was to use

standard UNIX processes. For example, initial sources, i.e., files, can be created by UNIX shell processes such as *ps*. These data sources can then be transformed using the UNIX utility *awk*, sorted using UNIX *sort* and merged using UNIX *cat*. Our program was written so that the functionality of other UNIX utilities could be incorporated easily.

Plumber has features, thanks to TICS, not easily provided by existing techniques such as UNIX shell scripts. These features include allowing the user to incrementally create and examine the results of specifying sources and transformations of data and providing the user with the power to easily modify his or her decisions. TICS' intelligent backtracking mechanism allows the user to modify both the data sources and the transformational plumbing. Because the intermediate results of operations are stored in files instead of being *piped* directly between processes, only those operations affected need be redone.

The result of a *Plumber* session, i.e., data source(s) plus transformation(s), can be viewed as a customized report. We will illustrate *Plumber* by describing a session that creates a report containing the current date followed by the list of an individual's running processes. The UNIX utilities that will be used in the scenario are `date` and `ps` to generate the sources, and `awk` and `cat` to transform and output the data.

The following program demonstrates TICS' ability to compose existing general-purpose processes in a user-oriented way. For each report a `create` goal is generated. Each `create` goal establishes a data source and that data's transforma-

tion. *Plumber* allows the user, via `manual or_mode` menu selections, to incrementally specify this source-transformation network. The data source either comes from an existing file or is created by generating the data from a UNIX utility. Each transformation is accomplished by a `translate` subtask. To utilize existing UNIX utilities and to display intermediate results, a few general purpose processes were developed, `show_file`, `user_atom`, `glue`, `t_file` and `do_shell`. These processes are described below.

```
$backtrack(manual).
$deduction(automatic).
```

Plumber creates the source and transformational plumbing, i.e., network of external processes, based upon the user's `manual or_mode` menu selections. However, these external processes require user-input, via the crt, before they will allow the data to flow through them. The `crt` gate is declared before the `select` gate to allow the crt-gated user-interactive processes to be started before the network of sources and transformations, established by `manual or_mode` goals, is completed. The user can thus incrementally create the plumbing and observe the flow of the data through the network. If the `select` gate were specified first, as in the previous *Car Buying* example, all the `or_mode` goals in the `select` gate would have to be solved before any crt-gated tasks could be started:

```
$gate(crt, 3, automatic).
$gate(select, 1, automatic).
```

The user can elect to do one or more independent *Plumber* reports. Each report consists of a network of sources and transformations:

```
$pred(report(Input, Output), [
    clauses([
     clause((report(Input, Output) :-
       create(Input, Output), report(Ninput, Noutput)), 2, 1,
       ['multiple interactive reports'], 1),
     clause((report(Input, Output) :-
       create(Input, Output)), 3, 1,
       ['single interactive report'], 1),
     clause((report(Input, Output)), 1, 1, ['end reports'], 1)
    ]),
    pdesc(['report', ' input ', Input, ' output ', Output], 1),
```

When the initial goal is solved the delayed side-effect process `r_temp` will be executed. This process will remove all temporary files by simply issuing a UNIX `rm tmp*` command. Temporary files are created by `t_file` as described below:

```
    side_effect(delayed, r_temp, []),
    or_mode(manual, 1)
]).
```

The `create` predicate establishes a data source. The source can be either an existing file or a UNIX shell command:

```
$pred(create(Input, Output), [
    clauses([
     clause((create(Input, Output) :-
       /* user enters the name of the data source file */
       user_atom('old filename ', Input),
       translate(Input, Output)), 4, 1,
       ['start with existing File ', Input], 1),
     clause((create(Input, Output) :-
       /* user enters a shell command to create data */
       user_atom('shell command', Command), t_file(File),
       do_shell(command(Command), input(none),
         output(File), Input),
       translate(Input, Output)), 2, 1,
       ['start with shell-created file ', Input], 1)
    ]),
    pdesc(['starting data'], 1),
    or_mode(manual, 4)
]).
```

The `translate` predicate establishes a transformation operation. The data can be edited, sorted or merged using the UNIX utilities *awk*, *sort* or *cat* respectively. Alternatively the series of transformations can be terminated with the data optionally being output to a file. *Plumber*'s framework is flexible; other UNIX utilities can easily be incorporated to provide additional transformations:

```
$pred(translate(Input, Output), [
    clauses([
     clause((translate(Input, Output) :-
       /* user enters the desired UNIX awk specification  */
       user_atom('awk ', Command), awk(Input, Command, Done),
       translate(Done, Output)), 6, 2, ['awk of ', Input], 1),
     clause((translate(Input, Output) :-
       /* user enters the desired UNIX sort options  */
       user_atom('sort ', Command), sort(Input, Command, Done),
       translate(Done, Output)), 6, 2, ['sort of ', Input], 1),
     clause((translate(Input, Output) :-
       create(NI, NOutput), merge(Input, NOutput, Done),
       translate(Done, Output)), 5, 2,
       ['merge ', Input, ' with another file'], 1),
     clause((translate(Input, Output) :-
       show(Input), translate(Input, Output)), 4, 2,
       ['show ', Input], 1),
     clause((translate(Input, Output) :-
       /* user enters the output file's name */
       user_atom('output filename', Output),
       glue(['cp ', Input, ' ', Output], CL),
       do_shell(command(CL), input(none), output(none), Done)),
       2, 2, ['output file ', Input, ' to ', Output], 1),
     clause((translate(Input, Input)), 1, 2,
       ['end translate ', Input], 1)
    ]),
    pdesc(['translate ', Input, ' to ', Output], 1),
    or_mode(manual, 6)
]).
```

To perform a merge operation we use *cat* to combine the files. The order of the files is determined by the clause selected to solve the `merge` subtask:

```
$pred(merge(F1, F2, Done), [
    clauses([
     clause((merge(F1, F2, Done) :-
       t_file(Temp1), glue(['cat ', F1, ' ', F2], CL),
       do_shell(command(CL), input(none), output(Temp1), Done)),
       2, 2, ['first file ', F1, ' then second file ', F2], 1),
     clause((merge(F1, F2, Done) :-
       t_file(Temp1), glue(['cat ', F2, ' ', F1], CL),
       do_shell(command(CL), input(none), output(Temp1), Done)),
       2, 2, ['second file ', F2, ' then first file ', F1], 1)
    ]),
    pdesc(['merge files', F1, ' ', F2], 1),
    or_mode(manual, 5)
]).
```

Invoke *awk* via the shell to perform a user-specified transformation:

```
$pred(awk(Input, Command, Done), [
    clauses([
     clause((awk(Input, Command, Done) :-
       t_file(Temp1), glue(['awk ', '"', Command, '"'], CL),
       do_shell(command(CL), input(Input), output(Temp1), Done)),
       2, 1, ['do awk :', Command, ' result in ', Done], 1)
    ]),
    pdesc(['change ', Input, ' with awk ', Command, ' to ',
      Done], 1)
]).
```

Invoke *sort* via the shell to perform a user-specified transformation:

```
$pred(sort(Input, Command, Done), [
    clauses([
     clause((sort(Input, Command, Done) :-
      t_file(Temp1), glue(['sort ', Command], CL),
      do_shell(command(CL), input(Input), output(Temp1), Done)),
      2, 1, ['do sort :', Command, ' result in ', Done], 1)
    ]),
    pdesc(['change ', Input, ' with sort ', Command, ' to ',
      Done], 1)
]).
```

Invoke `show_file` to display a file. The file's display is preceded by the file's name:

```
$pred(show(File), [
    clauses([
     clause((show(File) :-
      show_file(title(File), data_file(File))), 2, 1,
      ['do show'], 1)
    ]),
    pdesc(['show ', File], 1)
]).
```

The `show_file` predicate waits until its second argument is instantiated to the name of a file. This file is then displayed in a TICS window, optionally preceded by a title supplied by its first variable:

```
$pred(show_file(A, B), [
    ptype(external_generator, 'show_file', 1),
    pdesc(['show file ', A], 1),
    gate(crt, 2)
]).
```

If `user_atom`'s second argument is uninstantiated, the process displays the value of its first argument to prompt the user to enter a value. The entered value is then written to `user_atom`'s second argument:

```
$pred(user_atom(A, B), [
    ptype(external_generator, 'user_atom', 1),
    pdesc(['prompt user: ', A, ' entered ', B], 1),
    gate(crt, 4)
]).
```

The parts of a command line are sometimes supplied by multiple concurrent processes. We must wait for these parts to become instantiated before we can compose the complete command. The `glue` process waits until all the terms in its first argument are instantiated and then writes the concatenation of their values as a single atom to its second variable:

```
$pred(glue(A, B), [
    ptype(external_generator, 'glue', 1),
    pdesc(['create string (glue) ', B], 1)
]).
```

If the file supplied to `t_file` as its argument does not already exist, `t_file` creates a unique temporary file:

```
$pred(t_file(A), [
    ptype(external_generator, 't_file', 1),
    pdesc(['a file ', A], 1)
]).
```

The `do_shell` process is used to execute a UNIX shell command. The predicate's first argument specifies the command. The second and third arguments, if not instantiated to `none`, specify the redirection of standard input and standard output respectively. The shell command is issued after the predicate's first three arguments are instantiated. When the shell command completes, `do_shell` instantiates its fourth argument.

```
$pred(do_shell(A, B, C, D), [
    ptype(external_generator, 'do_shell', 1),
    pdesc(['shell ', A, ' ', B, ' ', C], 1)
]).
```

The following scenario is used to generate a report containing the current date followed by the process identifiers, *pid*, of a particular user's running processes. The user invokes the deduction engine by entering the initial Horn clause:

```
?-report(A, B).
```

In response to the `report` menu:

```
1-  multiple interactive reports
2-  single interactive report
3-  end reports
```

the user enters 2 to start generating the report.

In response to the `create` menu:

```
1-  start with existing file
2-  start with shell-created file
```

the user enters 2 to create a source of data from a UNIX shell command. This selection results in both a `user_atom` process prompt, *shell command:*, appearing in a

process window and a `translate` menu being posted. The user elects to further specify the network before entering the shell command and so selects *awk* from `translate`'s menu:

```
1-  awk of <unbound>
2-  sort of <unbound>
3-  merge <unbound> with another file
4-  show <unbound>
5-  end translate
```

Note: <unbound> appears in the menu because `t_file` has not yet created the file and bound the variable `Input` to the file's name.

As a result of selecting *awk*, choice 1, another `user_atom` process is invoked and posts the prompt *awk:* in a process window. Another `translate` menu is then posted. The user specifies another *awk* translation, which similarly generates another user prompt and a third `translate` menu. Curious about the current intermediate result the user selects `show` from this menu. A graphical view of this intermediate network is shown in Figure 5-2. Circles represent data sources and boxes represent data transformations. Each figure contains the name of its operation followed by the subtasks it invokes.

A fourth `translate` menu is posted. Since the user desires to examine the current results he proceeds to answer the previous `user_atom` prompts rather than further specifying the network via the `translate` menu. Answering the prompts allows the processes to proceed and the data to flow.

The value 'ps -uag' is entered for the *shell command:* to generate a source file containing a listing of all active processes. The value '/grossman/' is entered for the

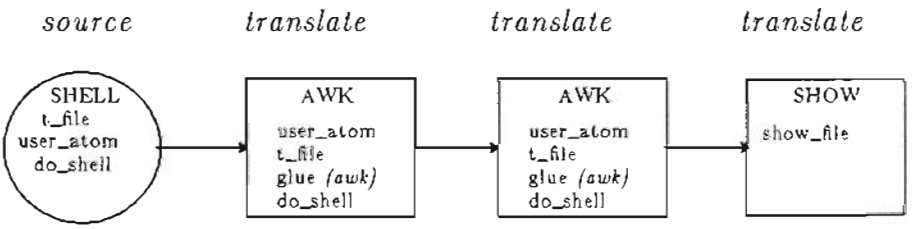


Figure 5-2. Intermediate *Plumber* Layout

first *awk* prompt to select only those process belonging to the user "grossman". For the second *awk* command the value entered is '{ print $ 2 }', which will project out the second column, i.e., the process identifier. As entries are made, processes that were previously invoked will be supplied data that they are waiting for. Data flows down the created plumbing from the source, generated by *ps*, through the two *awk* transformations and will then be displayed by the `show_file` process.

The user next specifies, via the `translate menu`, a *merge* transformation. The system invokes another `translate` menu to operate on the output of the merge. The other source of data for the merge will be provided later. The user specifies that the report should terminate with data going to an *output file*. A `user_atom` process prompts for the output file's name, and the user enters "my_report".

Before any output can occur the system needs to create and transform the other source of data to be merged. In response to the `create` menu, the user selects *start with shell-created file*. Again this selection results in both a `user_atom`

process prompt, *shell command:*, in a process window and a `translate` menu being posted. The user completes the network specification by selecting *end translate* and then enters 'date' for the shell command. As a result, a second source containing the current date is created. The source's data flows into the merge transformation where it is combined, via a UNIX *cat* process, with the list of process identifiers.

The resulting report is output to the file "my_report". By this time all external process have completed and terminated. A graphical view of the resulting network is shown in Figure 5-3. At this point the report is complete and TICS asks if additional solutions are desired. The user can choose to modify the plumbing, changing either the transformations or the data sources. In our scenario the user decides to change the report to contain the UNIX processes being executed by "maier". Since

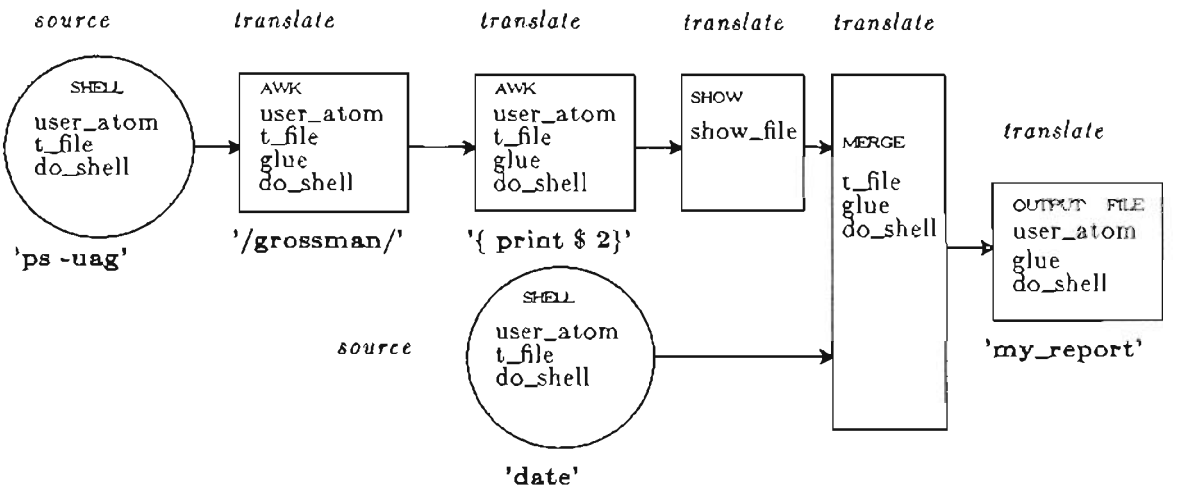


Figure 5-3. Completed *Plumber* Layout

backtracking has been set to manual, the user selects the subtask to be undone from the following menu:

```
1:
    prompt user: output filename entered 'my_report'.
2:
    prompt user: shell command entered 'date'.
3:
    prompt user: awk  entered '{ print $ 2 }'.
4:
    prompt user: awk  entered '/grossman/'.
5:
    prompt user: shell command entered 'ps -uag'.
    .
    .
    .
```

The menu choices are the leaf nodes, i.e., facts, in the deduction tree whose undoing can generate alternative solutions, as per our previous description of extended plan-based deduction. The choices include all the `external_generator` type processes. The process that supplied the first *awk* command, '/grossman/', is chosen by selecting entry number 4. The associated `user_atom`'s bindings are undone and the rerun process posts a new *awk*. prompt. In addition, TICS automatically resolves all predicates that used this data either directly or indirectly. Thus, the translate processes associated with the *awk: '{ print $ 2}', show, merge* and *file output* are run again. The processes associated with *ps* and *date* are not affected. When the user enters the new *awk* command, '/maier/', the data flows through this transformation and then through the rest of the network. The new list of process id's is displayed via *show* and the final data is output into the file "my_report". TICS once again prompts the user to see if additional solutions are desired.

If the user wants to get a more current listing of processes, he can request additional solutions. This action will invoke TICS' backtracking mechanism. He then can select the `do_shell` that had executed the *'ps -uag'* command, to be undone. This action will cause the *'ps -uag'* command and all processes that depended upon this command either directly or indirectly for data, to be rerun. The output file will then contain the new listing of the selected person's processes.

When the user decides to accept the solution, the delayed side-effect associated with the predicate `report` is invoked. This invocation initiates the execution of the process `r_temp`, which causes the removal of all temporary files created during the session. The source and transformational plumbing is no longer accessible after the user accepts a solution. TICS could be enhanced to generate a file containing the user's menu selections that resulted in a specific solution. This file could then be used by TICS to automatically regenerate the source and transformational plumbing.

5.3. Scheduling

The *Scheduler* example assigns professors and the courses they are to teach to classrooms. This task is invoked with a list of professors as its first argument and a list of time slots as its second argument. A time slot record contains name of classroom, hour of day, type of facility, professor-assigned and course-assigned fields. The last two fields will be specified with variables in the initial goal to indicate slots that are available for assignment. For example, we can use the following goal:

```
?- assign([mark, nora], [
    slot(room100, 10, lecture, A, AA),
    slot(room100, 1, lecture, B, BB),
    slot(room200, 10, seminar, C, CC),
]).
```

to invoke *Scheduler* for 2 professors, `mark` and `nora`. The goal lists 3 slots to which these professors may be assigned. The first `slot` specifies that room 100 of type lecture hall may be assigned at 10 o'clock.

Associated with each professor is an external file named "<professors_name>_courses". This file contains a list of records. Each record has three fields; the course name, desired time and required facility type. For example, a file named *mark_courses*, indicating that professor `mark` is to be scheduled to teach cs100 and cs101, contains:

```
cs100 11 seminar
cs101 10 lecture
```

The course cs100 requires a seminar room and `mark` desires to teach the course at 11:00.

The problem is to schedule all the "professor-course" pairs with the constraints that each time slot can only be assigned one "professor-course" pair. A "professor-course" pair must be assigned only to a classroom of the proper facility type. The professor's desired time requests will be satisfied if possible but an alternative time may be used.

The following program specifies the *Scheduler* task:

```
$gate(select, 1, automatic).

$pred(assign(Professors, ClassTimes), [
    clauses([
     clause((assign(Professors, ClassTimes) :-
       course_assign(start, Professors, ClassTimes)), 1, 1,
     ['assign professors to classes'], 1)
    ]),
    pdesc(['assignments: ', ClassTimes], 1)
]).
```

The course_assign procedure recursively generates a one_professor subtask for each professor in the list. The course_assign subtask will allow all one_professor processes to access the schedule at the same time. A one_professor process does not examine the schedule, ClassTimes, until its first argument is instantiated. To force a one_professor process to delay examining the schedule information until the previous one_professor process has modified the schedule we need only change the variable Start to Proceed This flexibility of operation is due to the design of the one_professor subtask and is described below:

```
$pred(course_assign(Start, Professors, ClassTimes), [
    clauses([
     clause((course_assign(Start, [], ClassTimes)), 1, 9,
       ['done'], 1),
     clause((course_assign(Start, [AP|R], ClassTimes) :-
       one_professor(Start, AP, ClassTimes, Proceed),
       course_assign(Start, R, ClassTimes)), 2, 1,
       ['assign one professor'], 1)
    ]),
    pdesc(['assign one professor'], 1)
]).
```

The one_professor process waits until its first argument is instantiated. The process then reads the name of its professor and accesses the professor's associated file. Assignments are made base upon the information in this file. This information specifies the course to be assigned, the professor's time preference and the type of room required. Assignments are made by instantiating variables, i.e., unassigned slots, in the classroom time slot list (argument three). When one_professor completes, it instantiates the variable Proceed to provide synchronization when the task is setup to operate in sequential mode:

```
$pred(one_professor(Start, P, C, Proceed), [
    ptype(external_generator, 'one_professor', 1),
    pdesc(['assign professor ', P], 1)
]).
```

Sequential versus concurrent access to the schedule information by the one_professor processes is similar to the difference between record locking versus optimistic concurrency control in the database world. Sequential operation ensures that each process will have sole access to the classroom time slot data during its time of operation. No conflicts will be generated because a process will know exactly what slots are safe to assign. A priority scheme is established by the order of the professors in the original goal. The nearer a professor is to the front of the list the greater his or her priority and thus the more likely to have the preferred time assigned.

When operating with concurrent access to the schedule information, the one_professor subtasks may generate conflicting assignments. These conflicts will cause TICS' intelligent backtracking to be invoked. The appropriate professor(s) process(es) will be undone and restarted. Eventually the system will, if possible, generate a non-conflicting set of assignments. We now illustrate the concurrent access version of *Scheduler* with the following external files for professors Becky, Nora and

Mark:

```
becky_courses:
 cs300 11 seminar
 cs301 10 lecture
nora_courses:
 cs200 1 seminar
 cs201 11 lecture
mark_courses:
 cs100 1 seminar
 cs101 10 lecture
```

A problem-solving session is invoked by submitting the following goal:

```
$deduction(automatic).
$backtrack(manual).
?- assign([becky, nora, mark], [
    slot(room100, 10, lecture, A, AA),
    slot(room100, 11, lecture, B, BB),
    slot(room100, 1, lecture, C, CC),
    slot(room200, 10, seminar, D, DD),
    slot(room200, 11, seminar, E, EE),
    slot(room200, 1, seminar, F, FF)
]).
```

The goal states that three professors are to be assigned to any of six classroom time slots. By setting the backtracking mode to manual the user will be given the choice of which professor(s) to undo in case there is a scheduling conflict.

In this particular scenario, the three `one_professor` processes accessed the schedule information before any one of them had made an assignment. Using this information the processes made assignments based upon their professor's external file. In this case the assignments resulted in two conflicts. The first conflict was that the name variable of the first slot, `AA`, was instantiated to both the values `becky` and `mark`. The second conflict was that the name variable of the last slot,

FF, was instantiated to both the values nora and mark. This set of conflicts could be removed by undoing the results, i.e., bindings, of either both of the one_professor subtasks for becky and nora or, of the one_professor subtask for mark. Backtracking was invoked and the user offered a choice of sets of procedures to be undone in the following menu:

```
1:
    assign professor 'becky'.
    assign professor 'nora'.
2:
    assign professor 'mark'.
```

The user selected set number 2, causing the bindings of the one_professor subtask for professor mark to be undone and the process rerun. The one_professor process then accessed the schedule, which still included the binding information from the one_professor subtasks for becky and nora, and made mark's course assignments. This assignment resulted in a conflict-free schedule and the original goal was successfully solved.

Chapter 6

Implementation

A TICS application consists of a special-purpose functional database and separate external processes that are used to solve evaluable predicates. The basic top-level architecture of TICS was presented in Chapter 4, and illustrated in Figure 4-1. We now discuss TICS in further detail.

Our prototype implementation was written in C++ and run under the 4.3 BSD UNIX operating system on a VAX[1]. When TICS is initially invoked, UNIX signal handlers are established to service <control-c> abort, abnormal child process termination, communication link failures and out-of-heap storage conditions. After the window system and interprocess communication mechanism are initialized, TICS polls for and services significant events. These events consist of keyboard input, messages from external processes and execution of the deduction engine. We now discuss, in more detail, each of these components of TICS.

6.1. Window Manager

The window manager maintains six windows called Transcript, State, Command, Process1, Process2 and Process3. Figure 6-1 depicts TICS' display. Keyboard input is handled by the window manager. All windows are active for output, but only one may be active for input. A special border is displayed around the window that is active for input. The State window is active for input in Figure 6-1.

[1]UNIX is a trademark of Bell Laboratories, VAX is a trademark of Digital Equipment Corporation

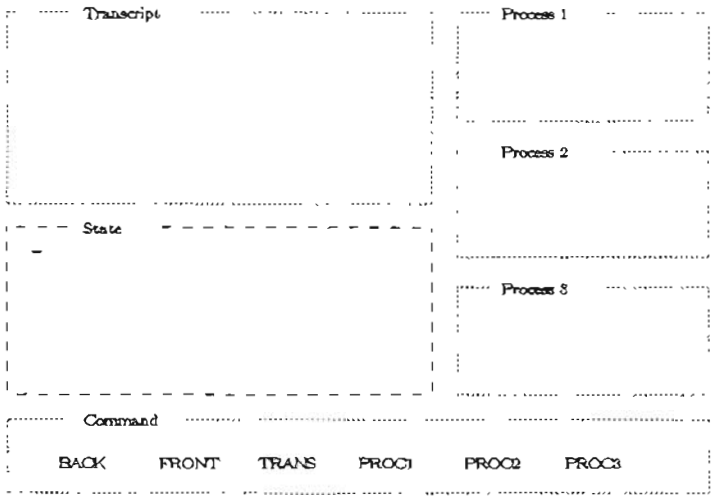


Figure 6-1. TICS' Window Manager

The Transcript window displays the occurrence of significant TICS events, e.g., compilation of Horn clauses, messages received from external processes, system errors. The State window allows the user to enter commands, e.g., compile a file of Horn clauses. The State window is also used to reply to questions from the deduction engine such as whether additional solutions are required. Both the Transcript and State windows record their displayed information in UNIX files. This information is used to provide a scrolling mechanism and a log of events that can later be analyzed. Process windows are allocated by external processes that need to interact with the user. An external process can display information on and receive user keyboard information from a window that it has allocated. If there are no Process windows available, a process will have its display messages printed in the Transcript

window by default.

The user can enter a window-manager command from the Command window. The window manager can be directed to select a new active window or, if Transcript or State is currently active, scroll the active window BACKwards or FRONTwards. The Command window becomes active whenever the user enters a <control-x> in any window. A command is selected by positioning the cursor using the <space> and <back space> keys and entering <cr>.

The window manager is called when keyboard data is entered. The manager buffers this input until a user message terminator occurs, i.e., <cr> or <esc>. Data input via the keyboard, while a Process window is active for input, is sent to the associated external process. If the Transcript or State window is active, the message is handled as a TICS command by the database access manager (DAM).

6.2. Database Access Manager (DAM)

TICS' command messages are sent to the DAM from either the user, via the keyboard as previously described, or by external processes. These messages are null-terminated byte sequences, i.e., UNIX character strings. The DAM interprets messages from the user and external processes in the same way. This common interpretation allows the designer to use the keyboard to simulate messages from external processes during system development.

To allow for user keyboard entry, we use only the alphanumeric characters. The first character of a message sent to the DAM specifies the function requested

and is sometimes followed by command specific information. The most common messages are:

Commands to manage the Horn clause base:

`b<filename>` *read and parse a file of Horn clauses*
`g<string>` *read and parse a string containing a Horn clause*
`c` *clear the Horn clause base*

Commands for utilizing the crt's Process windows:

`vg` *allocate an available window, if none available return an error*
`vr` *release any allocated window belonging to this process*
`d<string>` *display a string in the process's assigned window*

A process must initially identify itself and eventually a process should complete. This protocol is further described in the section on external processes. Commands for process identification, completion and backtracking control:

`i<process id>` *identification message*
`e<status>` *process completed with status (`f` or `s`)*
`f<variable id>` *fail goals instantiating the specified variable*

Commands to read, write and be notified of changes to logical variables:

`r<variable id>` *return the variable's current value*
`w<variable id>` `<value>` *write the value to the variable*
`n<variable id>` *return the variable's current value and notify of changes*

Note:

`<value>` is a *flattened* logical term. The structure of a logical term is described later in the section on data structures. A flattened term is a sequential character representation of a logical term and can be *reconstituted* into a logical term by parsing it.

<variable id> is the number assigned by TICS during deduction to represent a specific instance of one of a clause's variables. An external process is invoked with the identifiers of the variables in its environment. TICS numbering and handling of variables is described later in this chapter.

A message from the DAM is normally sent only in response to a message received by it, with the following exceptions. The first exception is when TICS needs to undo an evaluable predicate's active external procedure. If the process is of type external, TICS has the operating system kill the process. If the process is of type external_generator, TICS sends the process an abort message. When an abort message, <control-d>, is received, a process must immediately suspend or terminate. TICS will send an unsuspend message, a, if and when it requires the process to generate another solution. This technique allows an external_generator to preserve its local state information between invocations.

A second exception occurs when a process has asked to be notified about changes to a variable. When the DAM receives such a request, it immediately responds with the variable's current value. If, in the future, the variable's value should change, then TICS will send an unsolicited message containing the variable's id and value:

u <variable id> <value>.

Messages sent from the DAM, in response to a request, contain as their first byte either an ACK or NAK depending upon whether the the initial character of the message being responded to made sense. The second byte is the same as the first

byte of the request, i.e., the function requested. The third byte is either an `s` or `e`, depending upon whether the function was carried out successfully or if an error occurred. These bytes may be followed by data or error messages.

6.3. Horn Clause Parser

The parser, built using *lex* and *yacc*, handles Horn clause syntax in the format used by Clocksin and Mellish [Clocksin and Mellish 84]. When the DAM, via the window manager, receives a "compile file" command message or a "compile string" message, the parser is invoked. The parser translates most Horn clauses into predicate structures and stores these structures in TICS' Horn clause base. The base is implemented as a hash table with linked-list overflow. The hash code is generated from the clause head's functor and arity. Some types of clauses are not stored but instead invoke actions. When the parser translates a clause whose first term is `$deduction(Mode)`, the parser sets the deduction engine's forward mode of operation. If the term is `$backtrack(Mode)`, TICS' backtracking mode is similarly set. Clauses that begin with `$pred` contain a list of terms denoting predicate annotations. These terms are parsed and stored in a separate hash table. If `$pred` contains a `clauses` list, then the terms of this list are in turn parsed and added to the base. When a clause that begins with `?-` is parsed, the deduction engine is invoked to resolve the goal.

To illustrate the implementation we will now introduce a simple example called *Sample*. As was mentioned in Chapter 5, predicate annotations to control and manage freedom are optional. For the sake of simplicity, we keep these annotations

to a minimum in our example. The Horn clause specification for *Sample*, stored in a file of the same name, is as follows:

```
sample(R, S)  :- a(R), b(S), c(S), d(R), e(R, S).
a(i).
a(j).
b(j).
c(j).
e(T, T).
$deduction(automatic).
$backtrack(automatic).
$pred(d(Z), [ ptype(external_generator, dsample, 1) ]).
?- sample(X, Y).
```

To execute the *Sample* task the user directs TICS to read and parse the file by entering a command, `bSample`, in the State window. The parser is invoked and creates and inserts the entries for the first six clauses into the base. The resulting structures are shown in Figure 6-2.

The next two clauses cause the parser to set the deduction engine's global forward and backtrack modes to automatic. The `$pred` clause creates a predicate hash table entry for predicates with functor `d` and arity one. This entry indicates that its predicate is an external generator with an associated external procedure named `'dsample'` and with backtracking priority of 1. Parsing the final clause invokes the deduction engine to resolve the goal: `sample(X, Y)`.

6.4. Forward Deduction

The basic terminology and methodology of plan-based deduction was introduced in Chapter 4. In this section we describe the steps of forward deduction in more detail and then illustrate it with our program *Sample*.

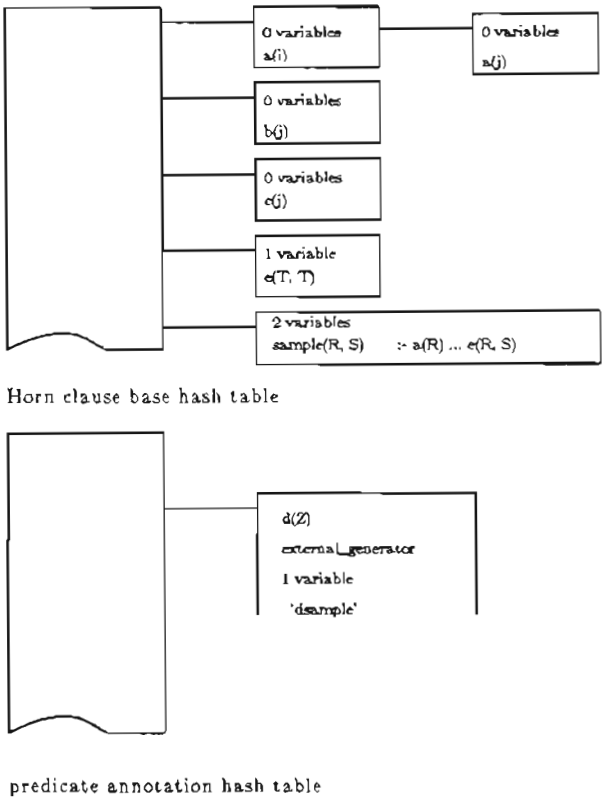


Figure 6-2. *Sample*'s Parser Created Structures

(1) TICS first creates a plan, i.e., deduction tree and associated constraint graph(s). The deduction tree's root represents the initial goal clause. This plan becomes the first entry in the list of viable plans called the *store*.

(2) While there are no unification conflicts and there are unprocessed goals, TICS services one subtask. This processing results in either the goal being solved or the subtask being inserted into a gate for later processing; this operation is described further below. If solving a subtask results in unification conflicts,

backtracking is invoked, otherwise we skip the next step.

(3) If TICS' intelligent backtracking, described in detail later, is able to generate a new plan without unification conflicts, then we return to the previous step and solve the new plan's goals. If a new plan cannot be generated, then TICS informs the user that a solution cannot be generated. Backtracking first attempts to generate the new plan from the current plan but if this fails, backtracking will use a plan from the store.

(4) At this point all open goals have been serviced initially, i.e., solved or gated. If there are no gated subtasks, TICS proceeds to the next step. If there are gated subtasks, then if the number of processes accessing the gate's resource is less than the gate's limit, TICS selects and processes one. If the number of processes accessing a non-empty gate's resource is not less than the limit, then TICS waits until one of these processes completes.

(5) If at any time there are unification conflicts, TICS invokes backtracking (step 3). TICS waits until all external processes have returned a completion status and then asks the user if the solution is acceptable.

(6) If the user desires another solution, then TICS creates artificial conflicts as described in Chapter 4, and returns to step 3 to generate another resolution. If the user accepts the solution, then TICS executes any delayed side-effects and completes with success.

Processing one Subtask

Processing depends upon whether the deduction engine's forward mode is automatic or manual. In automatic mode, the subtask to process is selected automatically from the list of open goals and one of the following steps is executed:

(1) If the goal's predicate uses a limited resource and has not yet been gated, the goal is inserted into the queue for the predicate's gate and no further processing is done at this time.

(2) If a goal has a non-empty potential, then if the goal's `or_mode` is `manual`, the user is prompted via a menu to select the clause to use from the goal's potential. Otherwise the system selects the highest priority clause. The selected clause is then removed from the goal's potential.

(3) If the goal can be solved by a built-in system predicate, e.g., external process or arithmetic operation, the system dynamically creates a node to solve the goal and, in the case of evaluable predicates, the associated external process is invoked or unsuspended.

(4) If the goal cannot be serviced by either of the first 3 steps then it is unsolvable and the goal is removed via backtracking.

In manual mode the steps are as follows :

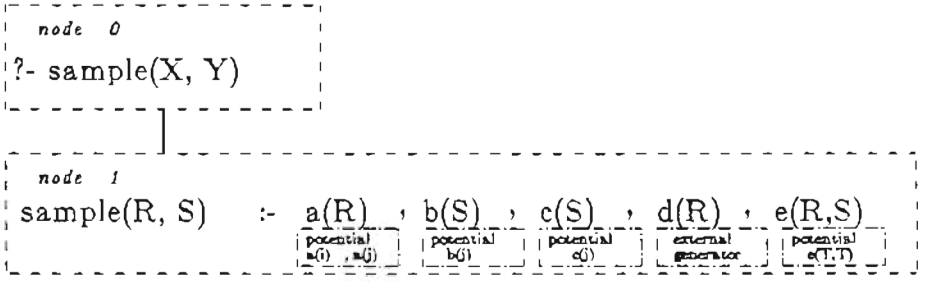
(1) The user selects the open goal to be solved.

(2) The user selects the clause from the goal's potential to solve the goal or specifies that a system predicate should be used.

For both modes of operation, a unique instance of a clause is used to solve the goal. This action causes the plan's deduction and constraint graphs to be updated. Any unification violations cause backtracking to be invoked.

Sample's Forward Deduction

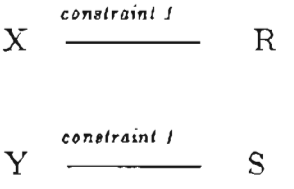
Let us consider again our *Sample* program introduced in Section 6-3. The initial goal, `sample(X, Y)`, causes the creation of node 0 in the deduction tree of the initial plan shown in Figure 6-3. The initial goal's potential, i.e., clause(s) that may



Boxes are nodes, i.e., clause instances

Vertical lines connect a goal with the node used to solve it

Deduction Tree



Lines represent unification imposed constraints

Constraint Graphs

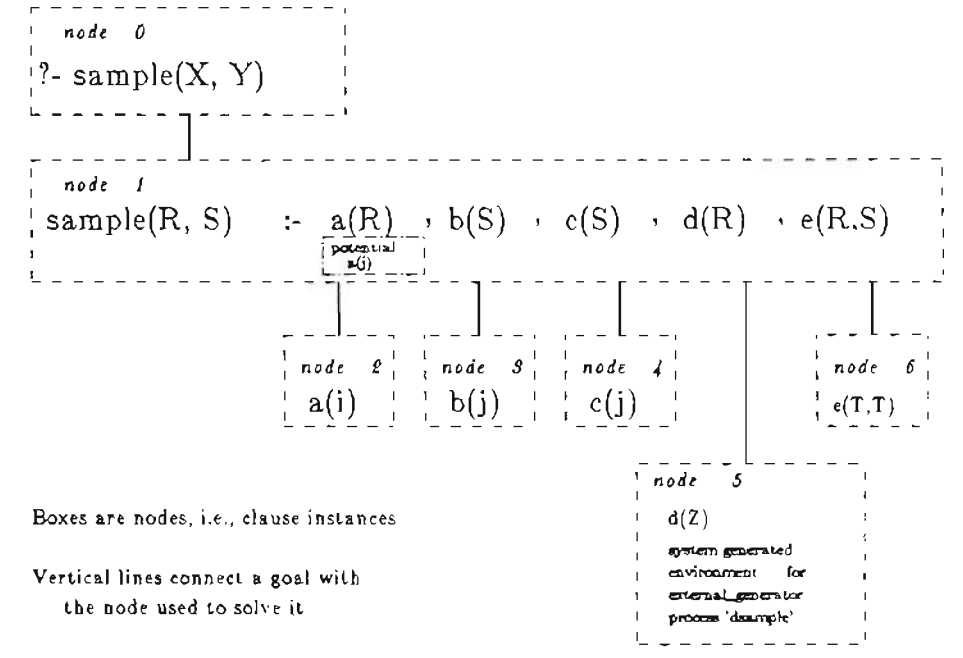
Figure 6-3. *Sample*'s Initial Plan With Two Nodes

be used to solve it, consists of the single clause:

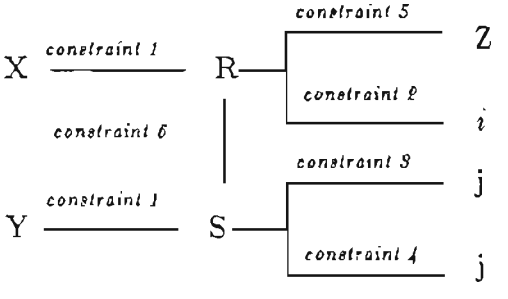
```
sample(R, S)  :- a(R), b(S), c(S), d(R), e(R, S).
```

The use of this clause results in the creation of node 1 in Figure 6-3. The unification constraints between terms is reflected in the plan's constraint graph(s). In the upper constraint graph of Figure 6-3, `X` and `R` represent their respective variables. The edge between the variables represents the unification constraint imposed by solving the initial goal by node 1. The unification of `Y` and `S` is similarly reflected in the lower constraint graph.

Node 1 establishes 5 new goals, `a(R)`, `b(S)`, `c(S)`, `d(R)`, and `e(R, S)`. The first of these goals is solved by the first of its two potential clauses, `a(i)`, establishing node 2 and adding a new constraint node, `i`, on an edge from `R`. These structures are shown in Figure 6-4. The second and third goals are solved by their single potential clauses, `b(j)` and `c(j)`, respectively. These actions add nodes 3 and 4 to the deduction tree, and two constraints between `S` and `j` to the constraint graph. Solving goal `d(R)`, an evaluable predicate, results in node 5 being generated to provide the environment of the external process "dsample". The process "dsample" is invoked and sends TICS a message requesting the value of its variable `Z`. The process receives in reply `i`, the value that the variable `Z` is bound to by the plan's unification constraints. The last goal is solved by its single potential, `e(T, T)`, creating node 6 and constraining `R` to `S`. There is now a unification conflict between the `j` terms of nodes 3 and 4, and the `i` term of node 2, causing backtracking to be invoked. This state of affairs is shown in Figure 6-4.



Deduction Tree



Lines represent unification imposed constraints

Constraint Graphs

Figure 6-4. *Sample*'s Initial Plan With Conflicts

6.5. Intelligent Backtracking

Conflicts occur either when a unsolvable goal is encountered or when there is a unification violation, i.e., non-unifiable logical terms are connected by edges in a constraint graph. The way extended plan-based deduction works is that forward deduction only operates on a conflict-free plan. When one or more conflicts occur, back-

tracking is invoked. As previously described, backtracking returns, if possible, a new plan that is created by removing a set of nodes that eliminates all conflicts from the old plan.

Unification violations are stored as a list of *conflict pairs*. Each conflict pair consists of two non-unifiable logical terms that are connected by edges in a constraint graph. For each of these conflict pairs, we determine all the ways possible to traverse the constraint graph between them, as discussed in the next section. These traversals generate a set of paths. A path is described by the set of deduction graph node numbers, representing the unification constraints used in one particular traversal. In our example, one path between the conflict pair `i` and `j` in Figure 6-4 consists of the nodes 2, 6 and 3. Removing any one node from a path's set of nodes will break the path. To remove a conflict, one node must be removed from every path for that conflict. To remove all conflicts one element must be removed from every path in the list. If we view a path as a logical term consisting of the disjunction of its nodes, then removing conflicts can be viewed as satisfying a logical formula whose terms are the conjunction of paths. Figure 6-4 contains a second path between `i` and `j` consisting of the nodes 2, 6 and 4. Restoring unifiability requires the removal of those nodes that satisfy the following logical formula:

$$(2 \cup 3 \cup 6) \cap (2 \cup 4 \cup 6)$$

We next remove redundant nodes from each path using Matwin and Piertrzykowski's *eliminate* function [Matwin and Pietrzykowski 85]. We seek to restore unifiability by destroying as little of our previous work as possible, i.e.,

removing the smallest amount of the deduction tree. A redundant node is one whose removal results in a deduction tree which is a subtree of a tree obtained by the removal of some other node in the path. For example, consider a path containing the two nodes "a" and "b". If "a" is a parent of node "b", then removing "a" would produce a subtree of the tree produced by removing "b". Thus, the node "a" is redundant. The eliminate function uses information in the deduction tree to remove redundancies. A node is removed if that node is an ancestor of another node in the path. Since none of the nodes in the paths in our example is redundant, the eliminate function has no effect. Had node 1 been included in either path, it would have been removed because it is the ancestor of some other node in the path.

Next we convert the conjunctions of disjunctions into a disjunction of conjunctions. In our example this operation results in the formula:

$$(2) \cup (2 \cap 4) \cup (2 \cap 6) \cup$$
$$(3 \cap 2) \cup (3 \cap 4) \cup (3 \cap 6) \cup$$
$$(6 \cap 2) \cup (6 \cap 4) \cup (6)$$

Removal of all the nodes belonging to any one of the disjunctive terms eliminates all unification conflict paths. Redundant terms — terms subsumed by another — are now eliminated. A term is subsumed if there is another term in the formula whose nodes form a subset of the subsumed term's nodes. For example, the term $(2 \cap 4)$ is subsumed by the term (2). Since unification can be restored by removing just the node 2 we need not consider removing both nodes 2 and 4. In our example this removal results in:

$$(2) \cup (3 \cap 4) \cup (6)$$

There is no point in removing a node if the parent goal has an empty potential, i.e., there are no more clauses that can be used to solve that goal. Therefore, we implement Matwin and Pietrzykowski's *pruning* function as described in Chapter 4. Pruning results in substituting for such a node its most recent ancestor node, which results in a plan whose unsolved goals have non-empty potentials. If this pruning fails to produce such a plan, then we discard this alternative, i.e., remove the conjunction of nodes of which it is a member. The resulting formula represents the **set** of sets of nodes that can be undone to restore unifiability. In our example, nodes 3 and 4 and 6 have only ancestor goals with empty potentials. Therefore, the only viable choice for removal is node 2, i.e., the **set** of sets of nodes is { {2} }.

To handle unsolvable goals and external process failures, we modified plan-based deduction to incorporate, for each plan, a list of nodes that must be removed. Each of these nodes is added to each set of backtracking alternatives, i.e., each element of the **set** of sets. In our example, if the external process "dsample" had terminated with failure, then node 5 would have been added to the single backtracking set resulting in the **set** of sets becoming { {2, 5} }. Since there is no other way to solve the goal `d(R)` or the goal `sample(X, Y)`, the deduction would fail.

Each of the backtracking alternatives can be used to generate a new plan. The original method for plan-based deduction creates all derivable new plans at once. We have modified this strategy to lazily generate new plans, one at a time, on demand. This strategy was adopted because people tend to solve problems by build-

ing upon current results, backing up only when required. When backing up, people usually return to the most recent state offering possible alternatives.

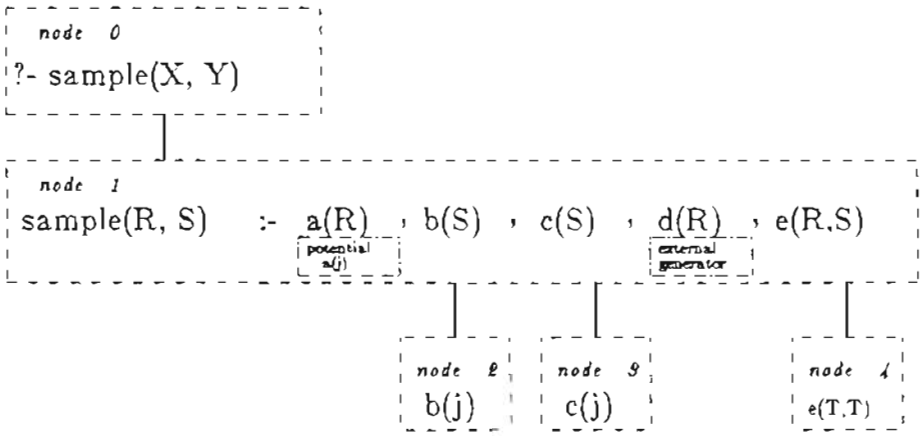
If backtrack mode is automatic, TICS selects the set of nodes to undo to create the new plan based upon priority. We have initially chosen to make a set's backtracking priority be the sum of the backtrack priorities of its elements. If the mode is manual, the user selects the set from a menu. A new plan is generated from the old plan by creating a new deduction tree and associated constraint graph(s) for all the nodes in the old plan, with the exception of those in the set of nodes to be undone.

External processes complicate the situation and force a major extension to plan-based deduction. The following external process validation procedure must be performed to ensure that causality constraints are not violated. As previously discussed, an evaluable predicate is solved by a system-generated node that represents the environment of the evaluable predicate's associated external process. All external processes that are part of the new plan must have the writes they issued, i.e., instantiations of logical variables, applied in the new plan. We must then ensure that causality constraints are not violated. Since processes are non-atomic, they can dynamically and incrementally examine variables, and may become causally dependent upon the values of those variables. The new plan's external processes must be checked to ensure that all the values they read are present, i.e., instantiated, in the new plan. If a value read by an external process is not present, TICS must remove the system-generated node that was used to solve the process's evaluable predicate.

In this case, another new plan is created and the entire external process validation procedure begun again. This procedure is repeated until a new plan's external processes are all found to be valid.

In our example, the first attempt at generating a new plan results in node 2 being removed. The resulting plan no longer has the variables `R` and `Z` constrained to `i`. Because the process "dsample" read the value `i`, it is no longer valid and another plan, with node 5 removed, is generated. Removal of node 5 does not cause the pruning of node 1 because `d(R)` is an evaluable predicate of type `external_generator` and thus can be re-solved, i.e., has potential. This plan, depicted in Figure 6-5, has no external processes and so its validation is trivial. TICS can now resume forward deduction. The goal `a(R)` will be solved by the fact `a(j)` and the external generator process "dsample" will be re-invoked. A solution will be achieved if "dsample" eventually terminates with success after reading the value of `Z`.

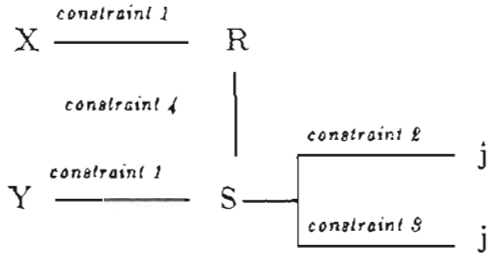
If the old plan has remaining elements in the **set** of sets of nodes, the plan is saved, because it may be used in the future to generate additional new plans. External processes cause additional complexity that may require us to update the old plan. All pending processes (as opposed to completed processes) must have their associated system-generated nodes marked for removal from the old plan. This action is necessary because we cannot be certain that the process will be in the same state if and when we return to the old plan. Consider the case where a plan has an external process that solicits information from the user. If backtracking is invoked



Boxes are nodes, i.e., clause instances

Vertical lines connect a goal with the node used to solve it

Deduction Tree



Lines represent unification imposed constraints

Constraint Graphs

Figure 6-5. *Sample*'s New Plan

with such a process and we do not remove the process from the old plan, then the following scenario might occur. The user, on the basis of the state of the new plan, supplies information to this process. The new plan is later discarded by backtracking and the old plan is again used to generate another plan. This latest plan would contain the pending process that was modified by the user in the context of a now defunct plan.

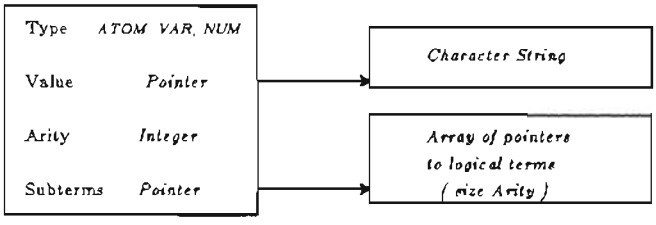
The system-generated node associated with an external process that completes with success is normally retained by the old plan. However, if the node's parent is a goal that has an inverse side-effect associated with it, then TICS removes the node because the responsibility for handling the inverse side-effect is passed to the new plan. If we did not take this action, the following scenario could occur. Goal "a" is specified to have an inverse side-effect procedure and is solved in the original plan by an external process represented by the system-generated node "n". A new plan is generated that also contains goal "a" and node "n". The new plan invokes backtracking and has node "n" removed. This action causes the goal's inverse procedure to be executed. TICS eventually uses the original plan to generate another new plan. This new plan also contains goal "a" and node "n". Backtracking is invoked and node "n" is removed from this new plan. As a result, the inverse procedure is again executed. There has been only one execution of the original procedure but there have been two executions of the inverse procedure. An alternative solution to this problem would be for TICS to globally track inverse side-effect goals. When an inverse side-effect procedure is executed, all plans that contained that goal would need to be updated. We chose, on the basis of simplicity, to implement the first solution in our prototype.

6.6. Data Structures

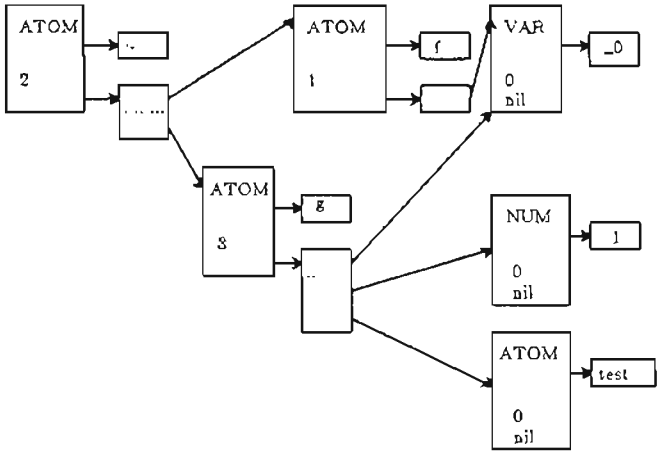
In this section we describe the data structures that TICS uses to implement the functions previously described. All logical terms are built out of the basic structure developed by Pase for his parser [Pase 86]. This structure consists of four fields and

is illustrated in Figure 6-6. The fields contain the term's type, value, arity and a pointer to either an array of pointers to subterms or, if the arity is 0, to nil. A term of type *VAR* is assigned a value equal to the variable's relative variable number as described below.

During forward deduction the current plan is developed by solving goals. Nodes are added to the plan's deduction tree and new unification constraints are reflected in the plan's constraint graphs. A node represents a clause instance and is described,



Logical Term Structure



f(A) :- g(A, 1, test).

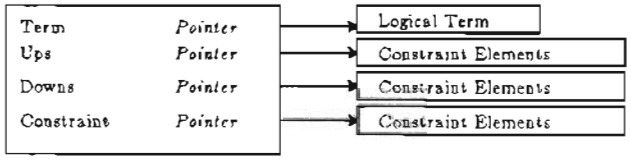
Figure 6-6. TICS' Logical Terms

in part, by pointing to the clause's entry in the hash table containing the Horn clause base. However, each clause instance must be assigned a unique set of logical variables.

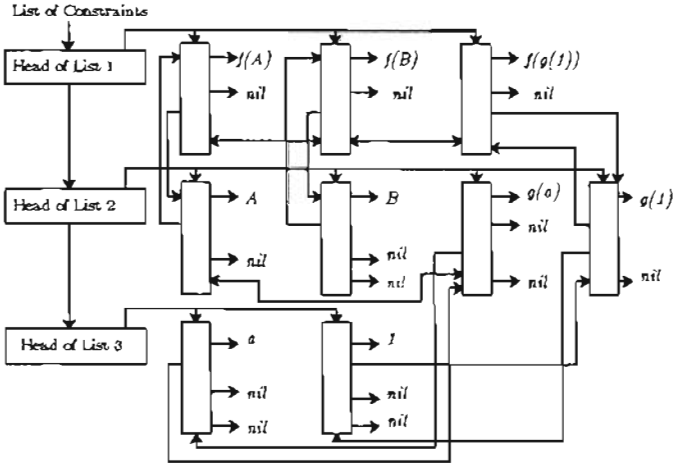
When the parser creates a clause's hash table entry it generates a structure with the variable subterms having numeric names. A clause's variables receive sequential numbers, the first variable being assigned 0. Variables with the same name receive the same number. TICS refers to these numbers as relative variable numbers. TICS assigns each node in a deduction tree a modifier number. The first node, node 0, is assigned a modifier of 0. Subsequent nodes have modifiers equal to their predecessor node's modifier plus the number of variables used in the predecessor node. The *id* for a node's variables is defined to be the sum of the node's modifier and the variable's relative number. In *Sample*, the variable `Y` of node 0 has a relative number of 1 and an id of 1. Node 1 is assigned the modifier 2 and its variable `S` has a relative number of 1 and an id of 3.

A variable's id is used to index an array of pointers to find the constraint graph structure, described below, that establishes the variable's value. When a new plan is generated, the nodes it initially copies from the old plan are assigned the same modifiers. This numbering scheme ensures that the ids for an external process's variables are the same in both the old and new plan. Thus, external processes that continue execution from one plan to the next will have the correct references for the variables in their environment.

The solving of goals result in unification constraints between logical terms that are reflected in a plan's associated constraint graphs. These graphs are composed of constraint elements that represent and link logical terms, as in Figure 6-7. This structure and its use is more fully described and illustrated by Cox [Cox 84]. A constraint element contains the following four fields. The first field is a pointer to the logical term it represents. The next field contains pointers to those constraint elements whose associated terms contain this constraint element's term as one of their



Constraint Element



Boxes represent constraint elements

Unification Constraints Structure

Figure 6-7. TICS' Constraint Graphs

subterms. The third field contains pointers to constraint elements whose associated terms are subterms of this constraint element's term. The next field contains pointers to constraint elements of terms that were unified with this constraint element's term.

When a unification constraint is established for a logical term, TICS creates a constraint element for that logical term if one does not already exist. If a new term is being constrained to a term with an existing constraint element, then its constraint element is added into the same constraint list, otherwise a new constraint list is created. The subterms of these logical term also have constraint elements associated with them and they too are added to the constraint lists. Figure 6-7 shows the structure that results from the following unifications:

```
1 - f(A) and f(B)
2 - A and g(a)
3 - f(B) and f(g(1))
```

The first unification initially generates constraint elements for the term `f(A)` and its subterm `A`. These constraint elements cause the creation of, and are added to, the first two constraint lists. Next, constraint elements are made for the term `f(B)` and its subterm `B`. These structures are linked to the constraint elements for the term `f(A)` and its subterm `A`, respectively. The rest of the structure of Figure 6-7 is generated by the remaining unifications.

The constraint pointer of a constraint element represents the unification created by a particular node in the deduction graph. Therefore, the constraint elements of subterms of terms being constrained do not have their pointers set. In

Figure 6-7, the element associated with A has its constraint pointer set to the element associated with g(a) but the element a does not have its constraint pointer set to the element associated with 1.

TICS can dynamically determine if a unification conflict has occurred by checking the new constraint elements against the other constraint elements in its associated constraint list. In the example described in Figure 6-7, unification 3 causes the creation of 3 constraint elements representing the term f(g(1)) and its subterm g(1) and g(1)'s subterm 1. When the constraint element for 1 is added to constraint list 3, a conflict with the constraint element for a is noted.

The graph of constraint elements can be used to determine the ways to restore unifiability. Paths between conflicting term's are determined by traversing the graph of constraint elements as per Cox [Cox 84]. This data is used to provide the intelligence in TICS' backtracking scheme. In the structure represented by Figure 6-7, there is only one way to traverse the structure between a and 1. This path requires using the constraint links created by the unifications 2, 1 and 3. Removing any of the nodes that created these unification constraints will remove the conflict.

To support the external-process validation, a procedure previously described, we incorporated the following additional structures into TICS. For each plan, TICS maintains a list of pending and a list of completed external processes. Each list has entries that record a process's id and pointers to the process's read and write lists. Read and write lists contain entries that record each reading or writing of a variable by the associated process. Each of these read and write records identifies the vari-

able accessed and the value that was read or written.

In Matwin and Pietrzykowski's implementation of plan-based deduction, many of the above structures were generated eagerly, i.e., before being required, to improve run-time efficiency [Matwin and Pietrzykowski 85]. We have modified their techniques to create all required structures dynamically upon demand. In this regard, TICS' extended plan-based deduction is both simpler and more space efficient. The additional run-time costs are offset by limiting the base to Horn clauses, eliminating the need for *ancestor resolution*: a technique used in plan-based deduction to achieve completeness for full first-order logic. TICS' evaluable predicates minimize the complexity of the deduction, thus further enhancing the efficiency and response of our system.

6.7. External Processes

As mentioned before, an external process is a UNIX process invoked by TICS to solve an evaluable predicate. Initially, a process needs to send an identifier message to TICS so that its process id, *pid*, can be associated with its *socket descriptor number*. If the process was designed to use the crt screen, the process must send a request to allocate a window, `vg` message, to the DAM.

Normally the next action a process takes is to parse its command line. The command line contains a flattened description of the process's local environment. The parse produces a logical term that may contain subterms containing logical variables. The value fields of these variable terms contain the variable's id, a number assigned by TICS as previously described. Different processes cannot access

the same variable because each process's environment is based on a unique deduction graph node and each node is assigned unique variables. However, there is nothing to prevent multiple processes from writing different values to variables constrained by unification. In such cases, TICS will invoke backtracking to remove the conflict.

A process can determine a variable's current value by sending a read message, `r<variable id>`, to the DAM. The DAM returns the unified value of all the logical terms to which the specified variable is constrained. In our *Sample* example, when the process "dsample" first read its variable `Z`, the DAM returned the value `i`; the unified value of the terms `X`, `R`, `Z` and `i`. When a process must ensure that a variable is instantiated, the process will normally first find out if some other procedure has already assigned the variable a value. How the process performs this action depends upon whether the process will, if the variable is unbound, write the variable or whether the process must wait for another predicate to issue the write.

If the process intends to supply a value if the variable is uninstantiated, the process issues a read request. If the variable is unbound, the process can then prompt the user, via a display message, to enter the required information. A write message is issued by the process to instantiate the variable. An example of this functionality is illustrated by the process `user_atom`, described in Section 5.2. If the process must wait until another predicate instantiates the variable, then it issues a notify request, `n<variable id>`, and will periodically check for an unsolicited-variable-changed message from TICS. Such a message will inform the process when and to what value the variable has become instantiated. The process

`bottom_line_constraint` in Section 5.1 uses this technique to obtain the values for the variables that contain the value of the trade-in, the cost of the new vehicle and the maximum amount that can be financed.

After a process has performed its required function, it normally will release any crt display space that it had allocated by sending a release window, `vr` message, to the DAM. The process can then terminate with an ending message indicating either success or failure, `es` or `ef`. A success status means that the subtask specified by the process's evaluable predicate has been satisfied. A failure status means that the process was unable to solve the subtask, i.e., the goal is unsolvable, and that TICS' backtracking mechanism should be invoked. An `external_generator` type process can, after successfully completing, suspend itself. The process can then use its internal state to provide another solution upon request.

TICS may cancel a process by either issuing an operating system kill command or sending a suspend/abort message. When so required, a process should be able to gracefully abort itself. We have also incorporated in all our external processes a user abort facility. In response to a process's prompt for information, a user can enter an abort request, `<esc><cr>`. When an abort request is received, the process sends a display message acknowledging the abort request and then terminates with failure. The user can thus force the system to try an alternative approach. This feature can be demonstrated in the *Car Buying* example, described in Section 5.1. Normally the `book_value` subtask is invoked with its `Method` variable instantiated to `low`, representing a dealer trade-in. If a user wanted to sell the old vehi-

cle privately, he or she could enter an abort request to the `book_value` subtask. The abort request results in the subtask terminating with failure and TICS generating, via backtracking, a new plan. This plan will solve the `do_tradein` subtask with its remaining potential, a clause that invokes `book_value` with its `Method` variable instantiated to `high`.

6.8. Implementation Considerations

The previous sections of this chapter have described, in detail, the different parts of TICS' design and implementation. Our prototype implementation combines these parts to provide a user-oriented task interaction and control system. To provide the functionality of extended plan-based deduction we utilize the following operating system features. The UNIX *signal* and *fork* facility are used to execute and control concurrent processes that implement external procedures for evaluable predicates. These processes communicate with TICS via UNIX *sockets*. We used the UNIX *curses* library to provide some of our window manager's functionality. The terminal independence of the *curses* routines allows us to run TICS applications on a wide variety of terminals.

The current TICS implementation requires approximately 5500 lines of C++ source code. This count does not include the code required for external function definition files, miscellaneous in-line functions, external processes and the Horn clause parser. The parser, based upon a *lex* and *yacc* specification provided by a colleague, Douglas Pase [Pase 86], required approximately 700 lines to specify and generated about 1800 lines of C code.

The example problem's external processes described in the last chapter were implemented in C++. These TICS applications had satisfactory performance and user-response times when the time-sharing VAX system was not heavily loaded. For the applications implemented so far, the computational complexity of TICS' extended plan-based deduction procedure does not seem unreasonable. We feel that even better performance could result from refining and tuning our prototype TICS to a specific operating system's hardware and software.

Chapter 7

Conclusions

In summary, TICS was found to provide a powerful problem-solving environment incorporating the following user-oriented features:

(1) Horn clause logic provides a clear and concise way to specify what constitutes an acceptable solution to a task. Logic's non-determinism supports multiple approaches to solving a task while indeterminism permits information to be supplied from any of a number of sources.

(2) Concurrent processes permit subtasks to be solved in any order consistent with the inherent nature of the task.

(3) The special-purpose functional database provides communication and supports synchronization between processes to allow subtasks to cooperate to solve inter-dependent subproblems.

(4) TICS tracking of dependencies provides an undo facility, via intelligent backtracking, that allows the system or user to modify answers with a minimum amount of lost work.

(5) TICS' delayed and inverse procedures provide two ways to cope with the issue of side-effects.

We noted the following designer-oriented features:

(1) The ability to logically decompose a task provides different levels of abstraction and detail, i.e., hierarchical modularity.

(2) The internals of external processes are hidden and isolated from the logic, providing the flexibility to use and re-use a wide variety of languages, tools and environments.

(3) The ability to decompose a task into multiple subtasks allows the designer to more easily separate the user-interface procedures from application procedures.

(4) The communication and synchronization mechanisms that support concurrent processes give the designer the potential to distribute the solving of subtasks to multiple processors. This distribution would require incorporating a means to allow the TICS process to communicate with, invoke and terminate external processes.

7.1. Observations

While designing and interacting with the various TICS application programs we made a number of observations. The ability to focus on subtasks allowed us to follow our own train of thought instead of the computer's. Repetitive tasks, such as giving demonstrations of the system, became much more interesting because there were so many new and different ways to solve a problem. A problem-solving session often held surprises, even for the system designer, as unexpected paths were explored. Backtracking was especially interesting because the complexity of subtask dependencies in many situations exceeded the designer's mental memory space. The user

could not always anticipate how the system would act but as the prototype's problems were fixed and our confidence and trust grew, this feature became a plus rather than a minus.

One example of this type of behavior occurred with the *Car Buying* task. After completing the initial task the user could request an alternative solution. If the user specified that the choice of color was to be changed (undone), then determining whether or not the style subtask had to also be re-done depended upon which had been done first: style or color selection. This unanticipated behavior arises because whichever subtask was last solved read the value instantiated by the subtask solved first. Reading this information allowed the second subtask to appropriately restrict the choices. A model is only available in certain colors and a color is only available in certain models. If the style selection was originally solved first, then it was independent of the color subtask. However if the color selection was solved first, then the style subtask read the information instantiated by color and became causally dependent upon the color subtask. Undoing the color subtask only affects the style subtask in the latter case.

Designing the applications described in Chapter 5 was a very positive experience. Each application took us less than one week to develop and test. While there is a significant effort required for a designer unfamiliar with TICS to learn the system, we believe that this overhead will be more than offset during an application's design, debug and maintenance phases. The time spent learning TICS should be less than that required to implement TICS' user-oriented features with traditional

methods. Freedom from arbitrary constraints translates into an exponentially large number of ways events can happen. The down-side of incorporating concurrency into a system is that such systems are more difficult to design, test and debug than constrained linear versions. TICS' provisions for the synchronization and communication of processes removes many of the problems a designer faces in developing systems incorporating concurrency. We have started developing a library of standard functions to help a designer implement concurrent processes.

To implement scrolling, our window manager logs all information written to the Transcript and State windows in a file. This log has also proved invaluable in analyzing what events occurred and what happened during system test and debug. TICS was found to allow for the easy re-use of existing solutions. Our work with Raimund Ege demonstrated the ability of using the power of other programming methodologies within TICS, generating a combined system with the advantages of both [Grossman and Ege 87].

While designing systems we confirmed what TICS is not. In TICS the subtasks that do the actual solving are specified by evaluable predicates, with associated processes. There is a temptation to use TICS' normal — non-evaluable — logic predicates to perform explicit problem solving, e.g., ensure that facts and bindings generated by external processes are correct, rather than just the problem decomposition they were intended to do. However, such techniques run into difficulties because with and-parallelism we cannot impose an ordering on predicate evaluation. TICS cannot ensure that a specific predicate is the one that instantiates the variables.

Thus, we cannot mix normal logic and evaluable predicates to create the traditional "generate and test" paradigm. (In Prolog we can be sure when two predicates work together to generate and test, the predicate that is leftmost in the clause is the generator.) The gating mechanism was not designed for this purpose and cannot always provide adequate sequencing of predicates to accomplish this style of problem solving. TICS could be extended to support optional mode declarations; however the thrust of our current research is to develop a system to coordinate the interaction between external modules, i.e., UIMS dialogue management. In some cases we can use normal logic predicates to resolve subtasks, as we demonstrated with *Car Buying*'s `in_stock` subtask.

We found that sometimes TICS' users would like to have certain complex actions performed by the system automatically. These actions require that TICS incorporate domain-specific knowledge. For example, in our version of Knuth's Pascal program, the user (or the system if in automatic backtrack mode) may have requested that the value for population size, `N`, be undone. Realizing that this is not what he or she desired, the user would like to say "No" and have the system undo sample size, `M`, instead. This "No" command requires that the "population size" subtask instantiate its previous value for `N`, and cause the system to fail and re-do the "sample size" subtask. The ability to handle this particular domain-specific command could be built into the external processes by the designer. A more general approach to domain-specific commands is a question for future research.

7.2. Future Research

Given enough money and time we would like to fully incorporate the following features into TICS. Some of the hooks are already in the code, a polite way of saying we have not implemented everything we have specified. Stream communication and private terms are not yet operational. Streams require that we provide an external process with the ability to dynamically create variables. We still need to implement an efficient means to create storage and assign identifiers to such variables. We have not fully implemented the techniques to protect private type objects from being accessed directly by the logic because the only such objects we have encountered in our examples are external files.

We would like to expand the current explanation and help facility. However, much of the difficulty of using TICS came from having a small display area and a lot of information to show the user. To effectively convey this information, we envision running TICS on a workstation with high-resolution graphics and multi-windowing support.

To support domain-specific requests it would be nice to incorporate a macro command facility whereby a single event could invoke a sequence of events to occur. The sequence could be made dependent upon the current state of the system, e.g., bindings of variables, global deduction and backtrack modes, external process status. One area where this feature would be useful is to better control automatic backtracking. For the *Car Buying* program the designer knows that the maximum amount of financing is available through the dealer. If the maximum financing,

EMax, only slightly exceeds the amount needed and if the financing method, EMethod, is dealer, then undoing the arrange_financing subtask would have a low chance of providing a successful solution and would probably make the situation worse. Under these circumstances it would be beneficial for TICS to lessen the possibility of undoing arrange_financing.

To provide these features, we would need to incorporate into TICS a mechanism to define and respond to selected events. We could create a special command message that when received by the DAM declares such an event. We could also create a new system predicate that would allow the designer to specify these events in terms of the state of the system. To respond to these events we might provide control points in TICS' algorithms that would transfer control to a designer-written routine. This methodology would be similar to the *escape* mechanism provided by *lex* and *yacc* to provide for the incorporation of arbitrary functionality via designer implemented C functions. These procedures could check for and take action based upon current system state information. In our *Car Buying* example, the designer might specify that whenever backtracking were invoked the system should execute a specified procedure to determine the backtracking choice. This procedure would be written to make its decision based upon its examination of the logical variable bindings and the status of external processes.

Currently, many asynchronous events are handled at the same level as the rest of the application. Handling asynchronous inter-process messages and user keyboard input via traps or interrupts, separate from the rest of the code, would be much

cleaner and more efficient. We did not have the time or ambition to tackle the job of getting UNIX sockets, terminal i/o and child termination to work together correctly via signals. A real-time operating system that supports asynchronous system traps (AST), event flags and shared memory for inter-process synchronization and communication would provide a more suitable environment in which to implement TICS.

BIBLIOGRAPHY

[Archer, Conway and Schneider 84]
Archer, James E., Richard Conway and Fred B. Schneider, User Recovery and Reversal in Interactive Systems, *ACM Transactions on Programming Languages and Systems Vol. 6*, No. 1 (January 1984), pp. 1-19.

[Bailey 85]
Bailey, D., The University of Salford Lisp/Prolog System, *Software Practice and Experience Vol. 15*, No. 6 (June 1985), pp. 595-609.

[Bentley and Knuth 86]
Bentley, Jon and Donald Knuth, Literate Programming, *Communications of the ACM Vol. 29*, No. 5 (May 1986), pp. 364-369.

[Borgwardt 84]
Borgwardt, Peter, Parallel Prolog Using Stack Segments On Shared-Memory Multiprocessors, *Proceedings of the the IEEE Symposium on Logic Programming*, 1984.

[Broverman and Croft 85]
Broverman, C. A. and W. B. Croft, A Knowledge-Based Approach To Data Management For Intelligent User Interfaces, *Proceedings of VLDB 85*, Stockholm, 1985, pp. 96-104.

[Cardelli and Pike 85]
Cardelli, L. and R. Pike, Squeak: a Language for Communicating with Mice, *ACM SIGGRAPH'85 Vol. 19*, No. 3 (July 1985), pp. 199-204.

[Chang and Despain 85]
Chang, Jung-Herng and Alvin M. Despain, Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis, *1985 IEEE International Symposium on Logic Programming*, Boston, July 1985, pp. 10-21.

[Clark and McCabe 82]
Clark, K. L. and F. G. McCabe, PROLOG: A Language For Implementing Expert Systems, in *Intelligent Systems: Practice and Perspective (Machine Intelligence #10)*, J. E. Hayes, D. Michie and Y. H. Pao (ed.), John Wiley and Sons, 1982, pp. 455-475.

[Clark and Gregory 85]
Clark, Keith and Steve Gregory, Notes on the Implementation of PARLOG, *J. Logic Programming Vol. 2*, No. 1 (April 1985), pp. 17-42.

[Clocksin and Mellish 84]
Clocksin, W. F. and C. S. Mellish, *Programming in Prolog*, second edition Springer-Verlag, Berlin, 1984.

[Conery and Kibler 81]
Conery, John S. and Dennis F. Kibler, Parallel Interpretation of Logic

Programs, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, October 1981, pp. 163-170.

[Conery and Kibler 85]
Conery, John S. and Dennis F. Kibler, AND Parallelism and Non-determinism in Logic Programs, *New Generation Computing Vol. 3*, No. 1 (March 1985), pp. 43-70.

[Cox and Pietrzykowski 81]
Cox, Philip T. and Tomasz Pietrzykowski, Deduction Plans: A Basis for Intelligent Backtracking, *IEEE Transactions on Pattern Analysis and Machine Intelligence Vol. 3*, No. 1 (January 1981), pp. 52-65.

[Cox 84]
Cox, Philip T., Finding Backtrack Points For Intelligent Backtracking, in *Implementations of PROLOG*, J. A. Campbell (ed.), Ellis Horward Limited, Chichester, 1984, pp. 216-233.

[Croft and Lefkowitz 84]
Croft, W. B. and L. S. Lefkowitz, Task Support in an Office System, *ACM Transactions on Office Information Systems Vol. 2*, No. 3 (July 1984), pp. 197-212.

[Davis 82]
Davis, Ruth E., Runnable Specification As A Design Tool, in *Logic Programming*, K. L. Clark and S. -A. Tarnlund (ed.), Academic Press, London, 1982, pp. 141-149.

[Draper and Norman 84]
Draper, Stephen W. and Donald A. Norman, Software Engineering For User Interfaces, *Proceedings of the Seventh International Conference on Software Engineering*, Orlando, Florida, March 1984, pp. 214-220.

[Forsythe and Matwin 84]
Forsythe, Kenneth and Stanislaw Matwin, Implementation Strategies For Plan-Based Deduction, in *International Conference on Automated Deduction. Proceedings of the 7th conference (Napa, 1984) [Lecture Notes In Computer Science; 170]*, R. E. Shostak (ed.), Springer-Verlag, New York, 1984, pp. 426-443.

[Furukawa, Nakajima and Yonezawa 83]
Furukawa, K., R. Nakajima and A. Yonezawa, Modularization and Abstraction in Logic Programming, *New Generation Computing Vol. 1*, No. 2 (1983), pp. 169-177.

[Gray, Moffat and Boulay 85]
Gray, P. M. D., D. S. Moffat and J. B. H. du Boulay, Persistent Prolog: A Secondary Storage Manager for Prolog, *Persistence and Data Types Papers for the Appin Workshop*, University of Glasgow, Glasgow, August 1985, pp. 353-368.

[Green 85]
Green, Mark, The University of Alberta User Interface Management System,

ACM SIGGRAPH'85 San Francisco Vol. 19, No. 3 (July 1985), pp. 205-213.

[Grossman 85]
Grossman, Mark, Humanizer - A framework for implementing flexible human-machine interfaces, unpublished manuscript, Department of Computer Science & Engineering, Oregon Graduate Center, May 1985.

[Grossman and Ege 87]
Grossman, Mark and Raimund Ege, Logical Composition of Object-Oriented Interfaces, *accepted for OOPSLA'87*, Orlando, October 1987.

[Hu 82]
Hu, T. C., *Combinatorial Algorithms*, Addison Wesley, Reading, Massachusetts. 1982.

[Jacob 83]
Jacob, Robert J. K., Executable Specifications for a Human-Computer Interface, *Proceedings of the CHI 1983 Conference on Human Factors in Computer Systems*, December 1983, pp. 28-34.

[Kahn and MacQueen 77]
Kahn, Gilles and David B. MacQueen, Coroutines and Networks of Parallel Processes, *Information Processing 77*, 1977, pp. 993-998.

[Kieburtz and Nordstrom 85]
Kieburtz, Richard B. and Bengt Nordstrom, The Design of Apple - A Language For Modular Programs, *Computer Languages Vol. 10*, No. 1 (January 1985), pp. 1-22, Pergamon Press Ltd.

[Kirkpatrick, Gelatt and Vecchi 83]
Kirkpatrick, S., C. D. Gelatt and M. P. Vecchi, Optimization by Simulated Annealing, *Science Vol. 220*, No. 4598 (1983), pp. 671-680.

[Komorowski 82]
Komorowski, H. J., QLOG - The Programming Environment for Prolog in LISP, in *Logic Programming*, K. L. Clark and S. A. Tarnlund (ed.), Academic Press, London, 1982, pp. 315-324.

[Kowalski 82]
Kowalski, R. A., Logic As A Computer Language, in *Logic Programming*, K. L. Clark and S. A. Tarnlund (ed.), Academic Press, London, 1982, pp. 3-16.

[Lamport 78]
Lamport, Leslie, Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM Vol. 21*, No. 7 (July 1978), pp. 558-565.

[Lientz and Swanson 81]
Lientz, B. P. and E. B. Swanson, Problems in Application Software Maintenance, *Communications of the ACM Vol. 24*, No. 11 (November 1981), pp. 763-769.

[Maier, Nordquist and Grossman 86]
Maier, David, Peter Nordquist and Mark Grossman, Displaying Database

Objects, *First International Conference on Expert Database Systems*, Charleston, April 1986, pp. 15-30.

[Matwin and Pietrzykowski 85]
Matwin, Stanislaw and Tomasz Pietrzykowski, Intelligent Backtracking in Plan-Based Deduction, *IEEE Transactions on Pattern Analysis and Machine Intelligence Vol. 7*, No. 6 (November 1985), pp. 682-692.

[Moffat and Gray 86]
Moffat, D. S. and P. M. D. Gray, Interfacing Prolog to a Persistent Data Store, *3rd International Conference on Logic Programming*, London, July 1986.

[Mycroft and O'Keefe 84]
Mycroft, A. and R. O'Keefe, A Polymorphic Type System for Prolog, *Artificial Intelligence Vol. 23*, No. 3 (August 1984), pp. 295-308.

[Nilsson 80]
Nilsson, N., *Principles of Artificial Intelligence*, Tioga Publishing Company, Palo Alto, California, 1980.

[Norman 83]
Norman, Donald A., Design Principles For Human-Computer Interfaces, *Proceedings of the CHI 1983 Conference on Human Factors in Computer Systems*, December 1983.

[Pase 86]
Pase, Douglas, *personal communication*, September 1986.

[Prywes and Pnueli 83]
Prywes, Noah S. and Amir Pnueli, Compilation of Nonprocedural Specifications into Computer Programs, *IEEE Transactions On Software Engineering Vol. 9*, No. 3 (May 1983), pp. 267-279.

[Reisner 81]
Reisner, Phyllis, Formal Grammar and Human Factors Design of an Interactive Graphics System, *IEEE Transactions On Software Engineering Vol. 7*, No. 2 (March 1981), pp. 229-240.

[Roach and Nickson 83]
Roach, J. W. and M. Nickson, Formal Specifications For Modeling And Developing Human/Computer Interfaces, *Proceedings of the CHI 1983 Conference on Human Factors in Computer Systems*, December 1983, pp. 35-39.

[Robinson and Sibert 82]
Robinson, J. A. and E. E. Sibert, LOGLISP: Motivation, Design and Implementation, in *Logic Programming*, K. L. Clark and S. A. Tarnlund (ed.), Academic Press, London, 1982, pp. 299-314.

[Shapiro and Takeuchi 83]
Shapiro, Ehud and A. Takeuchi, Object Oriented Programming In Concurrent Prolog, *New Generation Computing Vol. 1*, No. 1 (1983), pp. 25-48.

[Srivastava 86]
Srivastava, Aditya, The Explorer Prolog Toolkit, *TI Engineering Journal Vol. 8*, No. 1 (January-February 1986), pp. 93-107.

[Teitelman 75]
Teitelman, W., *INTERLISP Reference Manual*, Xerox PARC, Palo Alto, December 1975.

[Wadge and Ashcroft 85]
Wadge, William W. and Edward A. Ashcroft, *Lucid, the Dataflow Programming Language*, Academic Press, London, 1985.

[Walker 82]
Walker, Adrian, Automatic Generation Of Explanations Of Results From Knowledge Bases, Technical Report RJ3481 (41238), IBM Research Laboratory, San Jose, 1982.

[Wasserman 85]
Wasserman, A., Extending State Transition Diagrams for the Specification of Human-Computer Interaction, *IEEE Transactions On Software Engineering Vol. 11*, No. 8 (August 1985), pp. 699-713.

APPENDIX

System Predicates

The following summarizes the system predicates used to manage TICS' flexible deduction mechanism. A designer can use these predicates both to perform system debugging and to limit and guide the end-user's search strategy. TICS also incorporates system predicates to provide some arithmetic functions. Most of the system predicates have been illustrated in the example problems of Chapter 5.

Global Deduction Control

System predicates that cause, as a side-effect, changes in the deduction engine's global mode of operation are as follows:

`$deduction(Mode)`

This predicate specifies forward deduction control. `Mode` must be instantiated to either `automatic` or `manual`. In automatic mode, the selection of the subtasks to solve and clauses to use are made by the system on the basis of assigned priorities. This mode may be altered by `$pred` on a per predicate basis to allow selective control. In manual mode, usually used for single-step operation during debugging, the operator selects from a menu the subtask to solve and the clause to use.

`$backtrack(Mode)`

This predicate specifies backtrack deduction control. `Mode` must be

instantiated to either `automatic` or `manual`. When there is a unification failure in automatic mode, the system selects the set of nodes to remove from the deduction tree to restore unification on the basis of assigned priorities. In manual mode, the operator selects the set from a menu.

`$gate(Id, Limit, Mode)`

This predicate specifies a limited system resource. `Id` identifies the resource to be controlled. `Limit` specifies how many simultaneously executing subtasks can use this resource at any instant. `Mode` determines how gated subtasks are selected for solving. `Mode` must be instantiated to either `automatic` or `manual`. Predicates use the `$pred` system predicate, described below, to specify the limited resource they use and their priority for access to that resource. When the deduction engine tries to solve a subtask that uses a limited resource, it first puts that subtask into that resource's gate. When all subtasks have been serviced, i.e., solved or put into a gate, the deduction engine proceeds as follows. The gates are considered in the order they were specified. For each limited resource, subtasks are removed from the gate one at a time until the `Limit` is reached. If the `Limit` for that resource will not be exceeded by the subtasks in the gate then all the subtasks for this resource are removed from the gate and solved. If the `Limit` can be exceeded then only enough subtasks are selected, by the methods described below, to reach the limit. When a process that uses a limited resource completes, and there are subtasks in the gate for that resource, the deduction engine will select

one waiting subtask to solve. Selections are made according to the gate's Mode. In automatic mode, the system makes the selections on the basis of priority. In manual mode, the user selects from a priority-ordered menu. For example, `$gate(crt, 3, automatic)`, specifies that no more than 3 subtasks that use the crt can be active at once and that the system will automatically determine which subtasks to select from the gate.

Local Deduction Control and Predicate Annotation

Rules and facts that specify a task are stored together in the Horn clause base. Optionally, control and descriptive information about these rules and facts can be specified on a per-predicate basis by including assertions in the base of the form:

```
$pred(name(variable, ...), [Field, ... ])
```

The first argument of `$pred` is a structure that identifies the predicate being annotated. The functor, `name`, is an atom whose value is the name of the annotated predicate. The subterms of the `name` structure are variables; the number of these variables is equal to the arity of the annotated predicate. The second argument contains a list of one or more `Field` terms. Each `Field` specifies one annotation; their order does not matter. All `Fields` are optional, however, the inclusion of certain `Fields` may preclude the inclusion of other incompatible fields as noted below. In TICS, a number of different priorities are used to control and/or guide the operation. Priorities are always positive integers, the higher the number the higher the priority. `Field` can be any of the following :

`ptype(Type, Procedure, BP)`

The predicate's type is specified by this field. The designer uses `ptype` to declare an evaluable predicate. TICS solves an evaluable predicate by creating a new clause, i.e., node of the deduction graph, that provides an environment for the operating system process specified by `Procedure`. This process is invoked with a command line containing the new clause's environment, i.e., logical terms, which is described in Chapter 6. With this information the process can access the variables in its associated clause's environment. Backtracking priority, `BP`, is used when a unification conflict is detected, to determine the order of selection from the set of sets of clauses that can be undone to remove the conflict. This operation is described further in `clauses` below. `Type` must be instantiated to either `external` or `external_generator`. An `external` procedure can only be used once to solve an evaluable predicate. An `external_generator` procedure may be used to solve an evaluable predicate multiple times when required to by backtracking. A predicate not specified as evaluable is treated as a standard logical predicate. An evaluable predicate may not have a `clauses` field (see below). For example, the term, `ptype(external_generator, 'test', 1)`, specifies that the predicate is evaluable and will be solved by a new clause associated with the process *test*, i.e., a dynamic fact generator. This new clause is assigned a backtracking priority of 1.

pdesc(String, DisplayLevel)

This attribute is used to specify a predicate's description. String is a list of atoms and variable names that provide a natural language description of the predicate. The variable names in this list must be identical to the variable names in the $pred's name term. To describe a predicate, TICS prints the elements of the string replacing the variable names by the bindings of the corresponding arguments in the associated subtask at the time of printing. The DisplayLevel feature has not yet been implemented in TICS. DisplayLevel is to be used by the explain facility to provide different levels of detail to the user. The description will be printed when the desired display level is less than DisplayLevel. When no pdesc annotation is supplied, the actual predicate text is displayed by default. For example, consider the annotation:

```
$pred(person_city_state(A, B), [
    pdesc(['The preson ', A, ' resides in ', B], 1)
]).
```

This term would cause the following to be printed to describe the subtask person_address(C, D) if the variable C was instantiated to 'Jay Grossman' and the variable D was uninstantiated at the time:

```
The person 'Jay Grossman' resides in <unbound>.
```

If instead, the variable D were bound to the value 'Aloha, OR', the following would be output:

```
The person 'Jay Grossman' resides in 'Aloha, OR'.
```

`clauses([ ... ])`

This field contains a list of terms that describe clauses that can be used to solve the predicate. A `clause` term has the following format :

```
clause(Code, FP, BP, String, DisplayLevel)
```

`Code` is a Horn clause. `FP`, forward priority, is used to order the clauses used to solve a predicate. If clause selection is manual because either the `deduction` mode is set to `manual` or the predicate has an `or_mode manual` annotation (see below), then the user is presented with a menu with the goal's potential clauses listed in forward priority order. If clause selection is automatic because both the `deduction` mode is set to `automatic` and the predicate does not have an `or_mode manual` annotation, then the system automatically selects the highest forward priority procedure. Backward deduction priority, `BP`, is used to determine which predicate to undo during backtracking. If there is more than one element of the **set** of sets of goals that can be undone to restore unifiability then the sets are ordered by the sum of the backtracking priorities of each set's members. If `backtracking` mode has been set to `automatic`, then the system undoes the set with the lowest sum of backtracking priorities. If `backtracking` mode has been set to `manual`, the user selects the set from a priority ordered menu. `String` is a list of atoms and variable names that provide a natural language description of the clause. `String` and `DisplayLevel` work the same as for `pdesc`, providing a natural language format for the clause. The `clauses` annotation may not

appear for external and external_generator ptype evaluable predicates. For example, the following term:

```
clauses([
    clause( (c(V1) :- c1(V1)),2,1,['clause one'],1),
    clause( (c(V1) :- c2(V1)),1,2,['clause two'],1)
])
```

specifies two Horn clauses that can be used to solve the predicate being annotated. The first clause has the highest forward priority and the second clause has the highest backtracking priority. The last argument of the clause term specifies that the clause's DisplayLevel is 1. The DisplayLevel feature has not yet been implemented in TICS.

or_mode(Mode, Priority)

This field specifies the method for selecting the clause used to solve subtasks over this predicate. Mode is automatic for system clause selection on the basis of clause forward priority, or manual to provide selection by the user from a menu. Clause selection is implemented via a gate with an Id of select and a Limit of one. (If or_mode fields are specified, then a designer must explicitly provide a $gate term for select.) Priority is a number that is used to order system selection or menu presentation of the subtasks waiting for the select gate and is equivalent in function to the Priority argument of the gate predicate. Once an or_mode subtask is removed from the select gate it is solved as follows. If the subtask's or_mode is automatic mode, the system chooses a clause to solve the subtask on the basis of the forward priorities of the subtask's potential clauses. Otherwise the

user selects from a forward priority ordered menu containing the subtask's potential clauses. For example, a subtask described by a `$pred` annotation that includes the field `or_mode(manual, 2)`, will be inserted into the `select` gate with a priority of 2. When the subtask is removed from the gate the user will choose a clause to solve it from a menu containing the subtask's potential. The entries of this menu will be ordered by their forward priorities.

`gate(Id, Priority)`

This field indicates that the predicate utilizes a limited resource and its execution may be gated. `Id` is an atom that is used to identify the resource. `Priority` is a number that is used to order system selection or menu presentation. The default assumes that no limited resource is used. For example, consider a subtask whose associated `$pred` annotation contains the term, `gate(crt, 4)`. When that subtask is initially encountered by the deduction engine, it will be placed into the `crt` gate with a priority of 4.

`side_effect(Type, Procedure, ArgumentList)`

This field provides a subtask with the means to deal with side-effects. `Type` is either `delayed` or `inverse`. (The `inverse` side-effect can only be used with evaluable predicates.) The type `delayed` is used to prevent the occurrence of a side-effect until the problem is solved, i.e., an answer is committed to by the user. A subtask that is part of the final solution and whose predicate is of type`delayed` will have the process called `Procedure` executed. `Procedure` is used to cause the subtask's side-effect to take place after the

user's commitment. A subtask whose predicate is of type `inverse` will have the process called `Procedure` executed if and when the subtask is undone. `Procedure` is used to undo the predicate's side-effect. Both types invoke their process with a command line `Argument_list`. `Argument_list` is a list of logical terms. The variable names in this list must be identical to the variable names in the `$pred`'s name term. TICS replaces the names of the variables in this list with the current values of the corresponding argument in the associated subtask. For example, consider the following evaluable predicate annotation:

```
    $pred(a(A, B),[
        ptype(external, 'p', 1),
        side_effect(delayed, 'q', [B])
]).
```

TICS will solve a predicate, `a(F, G)`, with a new system generated clause, `a(H, I)` that provides an environment for the associated external process *p*. If this process has bound the variable I to the value 1 at the time the problem solution is accepted, then TICS will invoke the process *q* with a command line containing the numeric constant 1.

Arithmetic Predicates

System predicates that perform arithmetic operations are as follows:

`is`

The `is` operator is an infix operator. Its right-hand term must be an expression representing a numeric value, employing infix operators, `+`, `-`, `*`, `/`,

m representing plus, minus, times, division and modulo arithmetic operations. The right-hand term is evaluated and its value unified with the left-hand term.

< , > , <=, >=, ==, !=

These infix operators return true if their left-hand term is less than, greater than, less than or equal, greater than or equal, equal or not equal to their right-hand term, respectively. These operators require both left-hand and right-hand terms to be evaluable to numeric values.

Biographical Note

The author was born under the sign of Virgo on September 15th of the mid-year of the twentieth century, in the city of New York, borough of Brooklyn. At an early age he moved to Levittown, New York; the first suburban tract housing development. There he developed his distaste for unimaginative repetitive structures, e.g., dissertation formats. The author attended public school until 1968. He then attended Cornell University, majoring in engineering and was awarded a 2S, student draft deferment.

The next year the author transferred to S.U.N.Y. Stony Brook where he majored in psychology. After graduating in 1972 he joined the U.S. Coast Guard and underwent flight training with the U.S. Navy in Pensacola, Florida. He returned to the scene of his birth where he was stationed as a Coast Guard Aviator, flying helicopters for three years.

Upon his release, the author traveled until he obtained a job, house, wife and a child in the New England area. The work evolved from fixing terminals for Tektronix to diagnostic programming, writing device drivers and creating turnkey custom real-time computer systems for Digital Equipment Corporation. The author headed west in 1981 and a year later entered the University of California at San Diego. He worked as a teaching assistant and completed his M.S. in Computer Science in 1984. He then headed north to work on his Ph.D. at the Oregon Graduate Center.

The author is once again single and has a son, Jay, age 8, with whom he shares the summers. He is leaving Oregon to accept a faculty position at the University of Hawaii at Hilo.