A CONTROL STRUCTURE FOR TIME DEPENDENT REASONING

by
William J. Long and Thomas A. Russ*
Clinical Decision Making Group, Laboratory for Computer Science
Massachusetts Institute of Technology, Cambridge, Massachusetts

Abstract

In many domains it is important to reason about processes that change over time. Unfortunately, in many situations relevant data may not be available when the reasoning is done; there may be corrections to the data; or the state of the process may be changing. These problems are particularly evident in the reasoning involved in patient management. Physicians tend to reason about the patient state as a series of states. Therefore, a system attempting to capture their expertise must be able to find the appropriate time intervals in which to do the reasoning, handle incomplete data and handle changes to data even when the data relates to a state that has since changed.

We present a data-driven control structure for reasoning processes in a domain in which updating or changes can occur. This mechanism implements two abstractions for these processes: the abstraction of data from a continuous process and the abstraction of decision making in a static state context. We will illustrate the use of the system with an example from a medical expert system for patient assessment, but the techniques are also applicable to other domains such as business decision making with result tracking and sensor interpretation.

Introduction

Decisions in an intensive care setting often have to be made quickly even though information from such sources as laboratory tests (taking hours to process) are not available. When those results are received, they may still be pertinent but they refer to a previous patient state possibly modified by therapy.

A second problem concerns modeling change over time. Many processes, including many of those that provide patient data, evolve. Each new state is dependent on the previous state and current inputs. The obvious way to model such evolution is as a continuous process, but people tend to think in terms of states. That is, people consider a parameter to be low or high even though it is continuously varying or composed of separate values at time points. If a change has special significance, it is given a name and used in reasoning as if it were a static state: "The patient's blood pressure dropped this morning." These states are linked together by a more abstract notion of change. The key to such reasoning is picking the proper time intervals to divide a problem. This kind of behavior is reflected in a physician's summary of the patient's status. The patient is described as being in one state in the morning, but by afternoon having changed to another state.

Other Work

The problem we address is different from those addressed so far in temporal reasoning. Most recent research in that field has focused on reasoning about relationships between events occurring at different times [Allen, 1981a; Allen, 1981b; Bruce, 1972; Kahn and Gorry, 1977; McDermott, 1981]. The reasoning issues in domains where data arrives at different times has only been dealt with tangentially in expert systems to date.

To a limited extent MYCIN [Shortliffe, 1976], and the Digitalis Advisor [Swartout, 1977] have addressed this problem. MYCIN uses a complete recomputation strategy to correct previous information and the Digitalis Advisor uses a dependency directed updating strategy. In both cases, the changes are assumed to take place within one consultation session. Interestingly both programs use a static state assumption for most of the input data. For example, the Digitalis Advisor asks for the serum potassium concentration as if there were only one. VM [Fagan, 1980] is the closest in spirit to our work. It determines states from nearly continuous data, but is unable to update past assessments from newly received data about the past. Instead, data that is not current is either used as current or ignored as too old depending on how fast the particular parameter can change. Thus, the full problem of reasoning in a domain where data is received over time has not been addressed.

System Rationale

The motivation for this work is the need to take appropriate account of new data that pertains to past situations in which decisions have already been made. This often happens in the medical domain with laboratory data which takes hours to process, or when erroneous data is corrected. Treating such data as new data when it becomes available is wrong. Instead, we require that the program reexecute the reasoning in the time frame to which the data applies. Backtracking and withdrawing conclusions and then reexecuting the reasoning modules as if the data were available also has shortcomings because recommendations for actions in the past *cannot* be changed. Therefore, the reasoning routines must be able to distinguish between reasoning in the future, when both information (e.g., conclusions and diagnoses) and actions (e.g., drug therapy) can be changed and reasoning in the past, when only the information can be changed.

The recalculation and updating schemes discussed above are restricted to the current consultation. One reason is that they do not include any concept of *now*. *Now* is of consequence when calculations can produce action recommendations, because a re-calculation in the past must deal with the actual action as a given, even if it is judged to be incorrect in light of the new data.

* This research was supported in part by the National Institutes of Health Grant No. 1 P01 LM 03374–03 from the National Library of Medicine, in part by the Whitaker Health Sciences Fund, and in part by BRSG S07 RR 07047-17, awarded by the Biomedical Research Support Grant, Division of Research Resources, National Institutes of Health.

Thus, the control structure must:

- be able to determine appropriate intervals for reasoning and have the system enforce the state abstraction.
- be able to interpret the data as if interpreting a continuous process.
- allow changes to data in the past and control the process of changing the conclusions and even the reasoning intervals when appropriate.
- execute changes with a minimum of recalculation.
- support the distinction between the past and the future.

The system will be described using a procedural implementation of the reasoning units. The idea, however, is equally applicable to a rule or frame based system as long as there is a mechanism for processing raw data to determine appropriate intervals, explicitly representing data dependencies, and keeping a history of the data values.

System Description

The system has a relatively simple basic structure consisting of reasoning units, called *modules*, and the following kinds of variables:

1. **Point variables** represent the raw data for the program. They represent facts true at specific instants (e.g., laboratory values at specific times or actions such as the injection of a drug). Internally, a point variable is simply a list of data-time pairs. The part of this list pertinent to the time interval of interest is provided to the reasoning module during execution.
2. **Interval variables** support the state abstraction. As such they represent an interpretation of raw data over a time interval for which a constant interpretation is appropriate. Thus the values for an interval variable are represented as a set of non-overlapping time intervals with a single value for each interval. The system must be able to support both past and future values since programs are often called to reason about the future to do planning.
3. **Continuation variables** provide an interface between the continuous process abstraction and the state abstraction. They hold the process state information for continuous processes at specified time points. The problem they address is the breakdown of the continuous process abstraction when data is received out of chronological order. Our answer is to segment the process, remembering the state of the process in the appropriate continuation variable at the end of each time segment. Thus, the variable contains the information necessary to restart the process as if it were continuing from the previous interval. In this way a continuous process over a segment can be treated in the same way as other reasoning processes. In particular it can be reexecuted for any segment in which the input variable values have changed (including the state variable from the previous segment).

All system variables have associated with them the type, the history of values, and pointers to modules affecting them.

Modules are declared with the inputs and outputs explicitly listed as shown in figures 1 and 2. The only restriction on input and output variables is that each variable can appear in at most one module's output list, thus uniquely specifying the source when a value is needed. For each module the system creates a list of *processes* corresponding to the execution of the module over a time interval. Each process has an associated time interval to maintain the correspondence between the process and the variable values.

An Example

As an example of how such definitions could be constructed at both the data acquisition and reasoning levels, consider a (simplistic) pair of modules, one to interpret raw diastolic blood pressure data and one to evaluate blood pressure using interpreted diastolic and systolic values. In the example `raw_diastolic` is a point variable; `diastolic_cont` is a continuation variable; `diastolic`, `systolic`, and `bp_eval` are interval variables. The modules are called on the interval bounded by the system variables `begin_time` and `end_time`, with the current time set to `now`. The distinguished values `-infinity` and `infinity` are the system's end points. The purpose of `interpret_diastolic` (Figure 1) is to identify the interval over which the raw diastolic pressures are relatively constant and set the diastolic pressure for that interval to an appropriate value. `Evaluate_bp` (Figure 2) uses the abstract diastolic and systolic pressures and evaluates the blood pressure.

The values of the input variables are supplied to the code in local variables of the same name. They assume the interval from `begin_time` to `end_time` (except that the value of a continuation variable assumes the previous interval) unless times are explicitly stated. In this case `interpret_diastolic` needs the raw data from `begin_time` to `infinity` because changes or additions may cause the interval of stable values to be longer than before. `First_time` returns the time of the first data item (or `infinity`). `First_data` returns the corresponding data part. `Matching_time` returns the time of the first datum accepted by the predicate. The predicate `signif_diff` decides whether the difference between two values is significant enough to start a new interval. Since changes to the earlier values could have eliminated the difference between the previous interval and the current interval, the continuation variable is used to verify that a difference still exists. If not, the current execution is aborted with an explicit request to reexecute the module on the previous interval. When that is complete the module is executed again on the remaining part of this interval. After execution, the values for the output variables and the `end_time` of the interval are set from the local variables of the same name.

The algorithm used by `interpret_diastolic` identifies the points of significant change by making sure the change still holds at the beginning of the interval and by identifying the appropriate end of the interval using the same predicate. In this example the value for `diastolic` is the first

```
(defmodule interpret_diastolic
  ((raw_diastolic begin_time infinity) diastolic_cont) ;; input
  (diastolic diastolic_cont)                           ;; output
  (setq diastolic (first_data raw_diastolic))
  (cond ((= begin_time -infinity)     ;; prior to any data?
         (setq end_time (first_time raw_diastolic))
         (setq diastolic 'unknown))
        ((or (> (first_time raw_diastolic) begin_time)
             (not (signif_diff diastolic_cont diastolic)))
         ;; a gap in the data or no significant change
         (reexecute interpret_diastolic 'previous))
         ;; abort this process and reexamine the previous interval
        (t (setq end_time (matching_time
             '(lambda (x)(signif_diff diastolic x))
             raw_diastolic))))  ;; find time of first change
  (setq diastolic_cont diastolic))
```

Figure 1. Example of Raw Data Abstraction

```
(defmodule evaluate_bp (systolic diastolic) (bp_eval)
  (cond ((unknown? systolic diastolic)(setq bp_eval 'unknown))
        ((and (> (- systolic diastolic) 15)
          (between systolic 90 140)(between diastolic 60 90))
         (setq bp_eval 'normal))
        (t (setq bp_eval 'abnormal))))
```

Figure 2. Example of Constant Interval Code

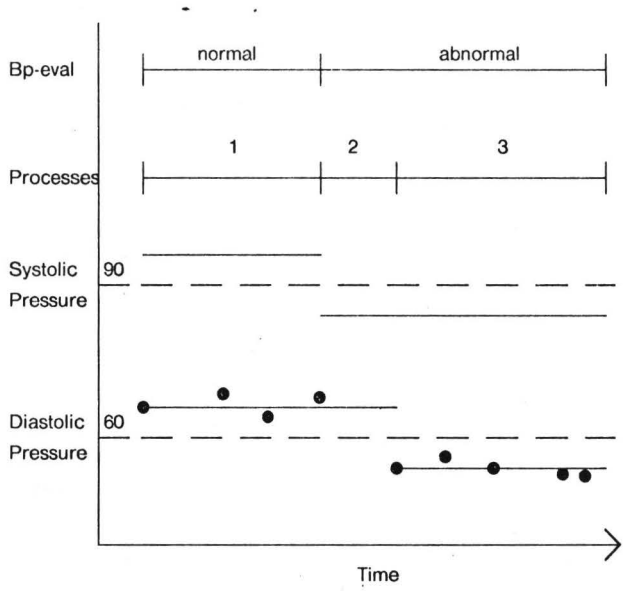

Figure 3. Time Behavior of Modules

raw_diastolic value in the interval. In a more demanding situation, the module might average the values, identify intervals over which the parameter is unstable, or provide information about the trend over an interval or between intervals.

The evaluate_bp module operates in a static state. When a request for bp_eval occurs in an interval for which the value does not exist, evaluate_bp is called with begin_time and end_time set for that interval. The diastolic and systolic values for evaluate_bp are provided by the system. If they do not exist, the modules having them as output are called; e.g., interpret_diastolic for the diastolic value. If the values are not constant over the desired interval, evaluate_bp is called on each successive sub-interval for which they are constant. This continuation is handled by the same mechanism that initially called evaluate_bp. If new data becomes available, the system would reexecute the module, changing the interval boundaries as necessary.

The result is reasoning code free from explicit handling of time dependencies.

Figure 3 illustrates the interaction of the two example modules. From the raw diastolic data, interpret_diastolic establishes two intervals, with pressures of 65 and 55 respectively. A similar module for interpreting the systolic pressure finds intervals with pressures 95 and 85, but the systolic pressure changes one data point before the diastolic. As a result, there are three intervals over which evaluate_bp is called. With a pressure of 95/65, bp_eval is normal. With pressures of 85/65 and 85/55, bp_eval is abnormal. Even though there are three processes for computing bp_eval, only two intervals result.

System Control

The system control is data driven. Processes to determine values are created when needed and values are propagated when the processes finish. If the value of a variable represents a change, the system reexecutes processes depending on the changed value, and so forth. This reexecution requires some care on the part of the control structure. The function that sets values must determine over what interval a new value actually constitutes a change. Otherwise, reexecution would always propagate to the end of the time space. To cover situations where changes may be arbitrarily small, it is possible to associate a *significant-change* predicate with a variable. In addition, a given variable may be an input for more than one module. When such variables change value, the modules that depend on the new value are queued to be executed. If the output variables change, the affected processes are queued. The queue is emptied in time order and data dependency order (to the extent the order can be determined from the module declarations). Even so, it is possible that a process will be executed more than once.

Since the conclusions of the reasoning modules are not valid until all changes are propagated, only the last execution's results should be displayed to the user. The system provides an output queuing mechanism to support this.

The effect of this control mechanism is that changes are always propagated through the record of what has happened. It is as if the continuous process of evaluating the input over time were rolled back to the point where a change was made and restarted at that point, except that the reasoning procedures are aware of the fact that they can not recommend new actions during time in the past. Only those modules are reexecuted that are needed to correct the execution history.

Conclusion

Providing system support for the data point and state abstractions allows us to model time-dependent processes in a natural way while keeping the domain code as free from computer housekeeping chores as possible. This simplification can be expected to ease the burden on knowledge engineers and enhance the reliability of expert systems.

References

[1] Allen, J. F., *A General Model of Action and Time*, Dept. of Computer Science, University of Rochester TR 97, November 1981.

[2] Allen, J. F., *Maintaining Knowledge about Temporal Intervals*, Dept. of Computer Science, University of Rochester TR 86, January 1981.

[3] Bruce, B. C., "A Model for Temporal References and its Application in a Question Answering Program," *Artificial Intelligence 3*, 1972.

[4] Fagan, L. M., *VM: Representing Time-Dependent Relations in a Medical Setting*, Ph.D. Thesis, Department of Computer Science, Stanford University, June 1980.

[5] Kahn, K. M. and G. A. Gorry, "Mechanizing Temporal Knowledge," *Artificial Intelligence 9*, 1977.

[6] McDermott, D. V., "A Temporal Logic for Reasoning About Processes and Plans," Computer Science Department, Yale University, RR 196, 1981.

[7] Shortliffe, E. H., *Computer Based Medical Consultations: MYCIN*, Elsevier North Holland, Inc., 1976.

[8] Swartout, W. R., *A Digitalis Therapy Advisor with Explanations*, Massachusetts Institute of Technology Laboratory for Computer Science, MIT/LCS/TR-176, February 1977.