An Architecture for Real–Time Qualitative Reasoning

Thomas P. Hamilton
United Technologies Research Center
Silver Lane
East Hartford, Connecticut, USA 06108
TPH@UTRC.UTC.COM

Abstract

Previous research on real–time qualitative reasoning has focused on making reasoning fast. Little work has been devoted to the more important issue of making real–time qualitative reasoning predictable. This paper describes an architecture for qualitative reasoning that addresses both speed and predictability. Extensions to the Qualitative Reasoning System (Hamilton 1988) to support predictable real–time performance are described.

1. Introduction

Development of qualitative, or model–based, reasoning systems is motivated by a number of well–documented benefits. These benefits include device independence of knowledge and reasoning techniques, clear definition of the scope of a system's competence, and the ability to diagnose faults with novel symptoms (Davis 1984). The importance of research on qualitative reasoning is substantiated by applications experience. In the HELIX program (Hamilton 1988), for example, it was discovered that experts – in this case veteran helicopter test pilots – do, in fact, base their diagnostic reasoning on an understanding of the underlying structure and function of the physical system.

Recently, research efforts have been aimed at applying qualitative reasoning to real–time problems. To date, these efforts have focused primarily on improving run time performance in qualitative reasoning (i.e., making qualitative reasoning fast). Hamilton (1988) uses hierarchical modeling to focus the diagnosis and prune the search space. Abbott (1988) explores the use of multiple representations and levels of abstraction to focus attention during reasoning. Doyle *et. al.* (1989) use a causal analysis to choose a subset of the predicted events to monitor from a potentially large number of available sensors.

As Stankovic (1988) points out, however, real–time computing is not equivalent to fast computing. "Rather than being fast (which is a relative term anyway), the most important property of a real–time system should be predictability" (Stankovic 1988). Optimizations such as those described above are clearly important to meeting stringent timing specifications. They do not, by themselves, guarantee that timing constraints will be met. Emphasis now must be shifted from making qualitative reasoning *fast* to making it *predictable* – guaranteeing that timing specifications will be met in real–time environments.

If the emphasis in real–time AI should be on predictability rather than speed, what are the barriers to building predictable AI systems? Fundamentally, the answer is complexity. Tasks typically assigned to AI programs are complex ones. As a result, the amount of processing that might be done to produce the best answer is frequently greater than that which can be done in the time available. Furthermore, the

primary mechanism for dealing with complexity in AI systems – search – is inherently unpredictable. The next section describes approaches to building predictable real–time AI systems.

2. Approaches to Real–Time AI

The goal of real–time AI is to get the best answer possible in the time available. Approaches to achieving real–time performance with AI systems may be broadly classified into three categories: Re–engineering, performance engineering, and time–constrained search.

2.1 Re–Engineering

The goal of re–engineering an AI system is to reproduce its functionality using a more traditional algorithmic approach. The benefit of using AI techniques comes from the understanding of the problem gained through rapid prototyping. Re–engineering the system using more traditional software techniques supplies the predictability required for guaranteeing real–time constraints are met. In this sense, the AI techniques may be considered a systems analysis tool.

2.2 Performance Engineering

Performance engineering on a knowledge base is a second method for achieving real–time performance in an AI system. The objective of this approach is to guarantee that the worst–case performance on a particular knowledge base satisfies timing requirements. This involves carefully analyzing and partitioning the knowledge base to bound the search process. For example, in a rule–based system, knowledge of the maximum number of rules that could fire and the time required to fire a rule can be used to determine the system's worst–case performance.

2.3 Time–Constrained Inference

A third method for achieving real–time performance in an AI system is to bound the search process by directly imposing time constraints. Unfortunately, many problem–solving approaches are structured in such a way that partial solutions present little or no useful information to the user. That is, the result of reasoning is only of value if the search runs to completion.

Reasoning algorithms that offer the greatest benefit in dynamic real–time environments are those that are able to return a useful partial solution even if unable to run to completion. Thus, desirable characteristics of a real–time problem–solving algorithm are:

1) partial solutions to the problem represent useful approximate answers,
2) processing may be interrupted and partial solutions returned when a time limit is reached, and
3) if additional processing time becomes available, it should be possible to continue the unfinished search to further refine the answer.

The approach to achieving real–time qualitative reasoning described here is based on the time–constrained inferencing approach. The remainder of this paper presents a brief overview of the Qualitative Reasoning System (QRS), extensions to QRS to support real–time operation, and a real–time testbed for QRS.

3. QRS Overview

The Qualitative Reasoning System (QRS) is a software system designed to support the development of qualitative reasoning applications. QRS consists of two primary components - a Qualitative Model Builder and a Qualitative Reasoner. A brief overview of the capabilities

of QRS is presented below. A detailed description of QRS may be found in Hamilton (1988).

3.1 Qualitative Model Builder

The Qualitative Model Builder provides an interface for construction, storage, and testing of qualitative models. Qualitative models in QRS are of two basic types: *Elementary Models*, representing devices without substructure, and *Compound Models*, representing devices composed of components. Elementary models are defined using a series of menus for specifying the device's variables and constraints.

Compound models are constructed graphically with the model–building interface depicted in Figure 1. Icons corresponding to model prototypes are selected from the menu at the right and moved into the model–building window at the left. With the mouse, conduits are created specifying the interconnections between components. Using the Qualitative Model Builder in this way produces collections of hierarchical model representations that may stored into and retrieved from model libraries.

The Qualitative Model Builder also provides an interface to the reasoning algorithms in the Qualitative Reasoner. This allows interactive testing of elementary and compound models.

3.2 Qualitative Reasoner

The Qualitative Reasoner provides utilities for reasoning over a qualitative model. These utilities may be accessed in an interactive, menu–driven fashion or may be called directly by a problem–solving application. The Qualitative Reasoner operates on models to perform the following operations:

- State Generation - determining all possible assignments of qualitative values to variables that are consistent with a model and zero or more observations.
- Fault Detection - determining if observed values conflict with a model.
- Diagnosis - determining, using hierarchical constraint suspension (Davis

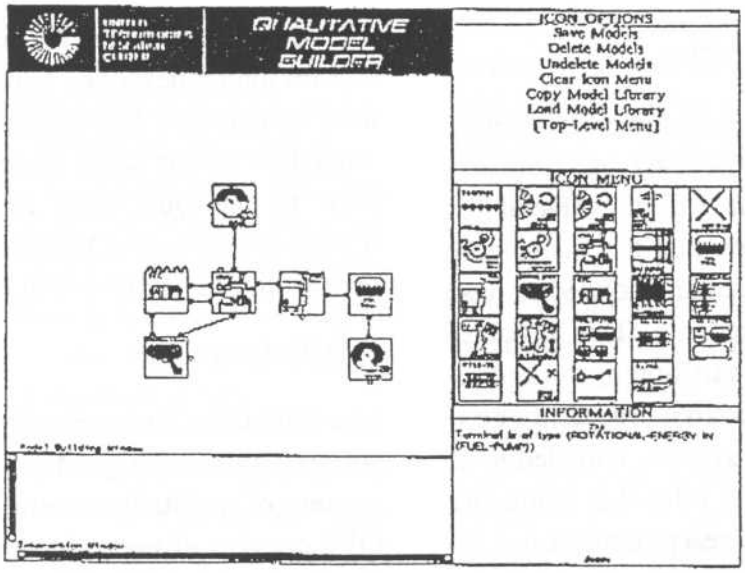

Figure 1. The QRS Qualitative Model Builder. The Qualitative Model Builder provides a graphical interface for building, testing, and modifying qualitative models.

1984), what component failures account for a detected fault.

- Sequential Diagnosis - interactive troubleshooting based on system–generated test recommendations (de Kleer and Williams 1987).

4. pQRS – Parallel QRS

The hierarchical approach to model representation and diagnosis used in QRS is ideally suited to parallelization. As a diagnosis proceeds top–down through the model hierarchy, valid hypotheses are identified. Each valid hypothesis that contains substructure is expanded into a new set of finer–grained hypotheses. The evaluation of these new expanded hypotheses can be distributed to available processors and executed in parallel.

The architecture for parallel QRS, or pQRS, is shown in Figure 2. pQRS is made up of three components. The Diagnosis Manager accepts observations, detects faults, initiates fault isolation, and returns the diagnosis. The Hypothesis Scheduler maps hypotheses to be tested onto remote processors for execution. Hypothesis Servers, residing on remote processors, evaluate hypotheses and return the results. Each of these three components is discussed in turn.

4.1 The Diagnosis Manager

When the Diagnosis Manager receives a set of observations, the first step is to run the Fault Detector to determine if the observations are consistent with the normal behavior of the device. Observations are propagated through the constraints of the root model to determine if there exists a state consistent with the model and observations. If a state consistent with the model and observations can be generated, the observations are considered normal and the Diagnosis Manager returns this result.

If there is no state consistent with the root model and observations, then a fault is detected. The Diagnosis Manager then initiates fault isolation by creating hypotheses and submitting them to the Hypothesis Scheduler. Each hypothesis created by the Diagnosis Manager corresponds to a test whether the observations can be explained by a failure in one of the components of the root of the model hierarchy.

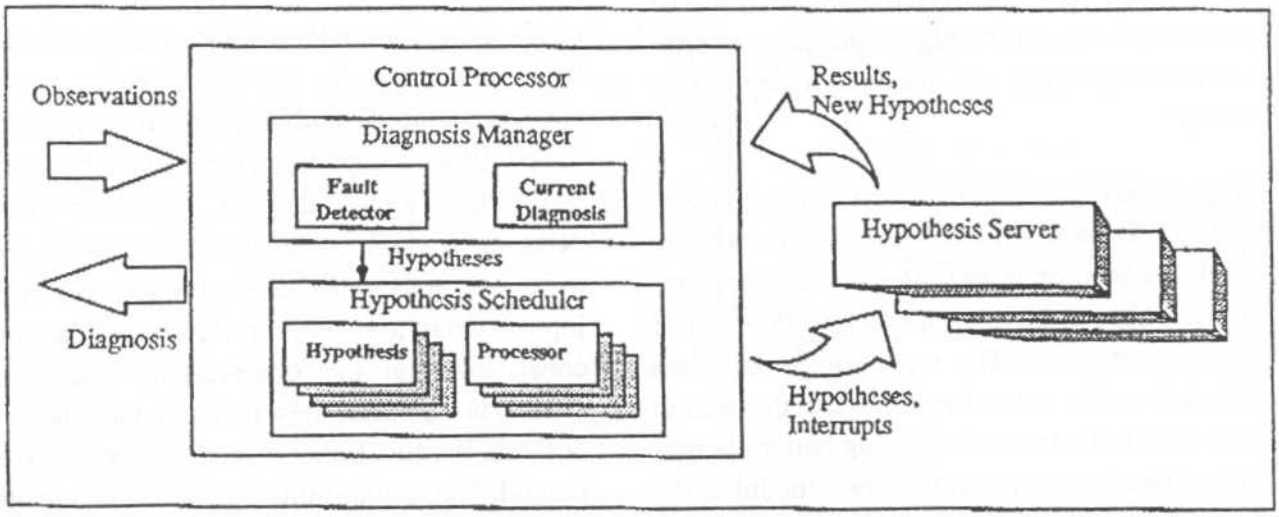

Figure 2. The pQRS Architecture. Hypotheses to be tested are distributed to available processors by the Hypothesis Scheduler to perform a parallel hierarchical diagnosis.

Each hypothesis consists of a *model view* and a set of observations. The model view specifies the level of detail with which to evaluate the hypothesis. The model view contains two lists: the first is the list of components assumed to be working; the second contains the components to suspend. Together, these lists determine which constraints are in the model when the hypothesis is evaluated. The observations are pairs, each consisting of a model parameter and its measured qualitative value.

4.2 The Hypothesis Scheduler

The Hypothesis Scheduler's duty is to maintain two queues. The first is a Processor Queue containing the processors currently available to evaluate a hypothesis. The second queue, called the Hypothesis Queue, contains the hypotheses waiting to be evaluated.

The type of search performed by pQRS can be controlled conveniently by assigning priorities to the hypotheses in the Hypothesis Queue. Assigning priorities according to a hypothesis' position in the hierarchy produces a standard breadth– or depth–first search. For example, if each hypothesis is given a lower priority than its parent, a breadth–first parallel search will be performed. Assigning higher priorities to children than parents will produce a depth–first search.

As pointed out by Lesser *et. al.* (1988), it is possible to avoid computation costs by avoiding work on low–certainty alternative interpretations. This can have an impact on the type of search performed. The more conservative the scheduler, the more breadth–first the search will tend to be to avoid missing better alternatives. The less conservative the scheduler, the more depth first the search will be to avoid expending computing resources on low–certainty alternatives. Other factors, such as the criticality of failures in different branches of the hierarchy, may also influence the type of search performed.

The Hypothesis Scheduler distributes hypotheses, one per processor in the processor queue. Each valid hypothesis may create additional, more–specific, hypotheses. These are added to the Hypothesis Queue with a priority determined by the search scheme. The highest priority process is created on the first available processor. That processor is removed from the Processor Queue; it gets enqueued again after its process completes.

4.3 Hypothesis Servers

A Hypothesis Server is a process, running on a remote processor, whose job is to evaluate hypotheses. Each Hypothesis Server performs the following duties:

- Accepts requests to test hypotheses
- Evaluates hypotheses
- Returns result
- Creates new hypotheses
- Keeps track of its availability and informs the processor scheduler

To evaluate a hypothesis, a Hypothesis Server needs to have a model instance for the device being monitored. This instance enables any single or multiple fault hypothesis to be evaluated. When a hypothesis is received, the specified model view is created. The model view determines which constraints of the model are to be reasoned over. The observations are then input to the state generation algorithm. If a state consistent with the observations and constraints in the model view is found, then the hypothesis is valid. Otherwise, the hypothesis is invalid. The result of this test is returned to the Diagnosis Manager and stored with the Current Diagnosis.

If a hypothesis is valid, the Hypothesis Server must also determine if additional, more–detailed hypotheses need to be created. Any valid hypothesis representing a failure in a component with substructure must be expanded. These expanded hypotheses are submitted to the Hypothesis Scheduler to be enqueued in the Hypothesis Queue.

4.4 Benefits of pQRS

Benefits of pQRS include the following:

- Improved response time,
- Flexible control of search,
- Ability to dynamically change priorities,
- Fault tolerance, and
- Support for time–constrained inference.

As discussed previously, fast response does not make pQRS a real–time system. What is still missing is predictability. The architecture for pQRS does, however, provide the foundation on which to build a predictable real–time qualitative reasoning system. Section 5 describes how the pQRS architecture can be extended to ensure predictable performance.

5. QRSt – Time–constrained QRS

The hierarchical representation of QRS is well suited to time–constrained inference. Each successive level in the hierarchy represents a finer–grained analysis. A partial solution for the QRS diagnostic process might correspond to isolating a fault down to the system or subsystem level. Such a partial solution, though less detailed than it might be given additional computing resources, can still be of great value in deciding how to respond to the failure.

Extensions to QRS to support time–constrained inferencing result in adding two components to the pQRS architecture. As shown in Figure 3, these components are the Time Manager and the Interrupt Manager. These components are described in the following two sections.

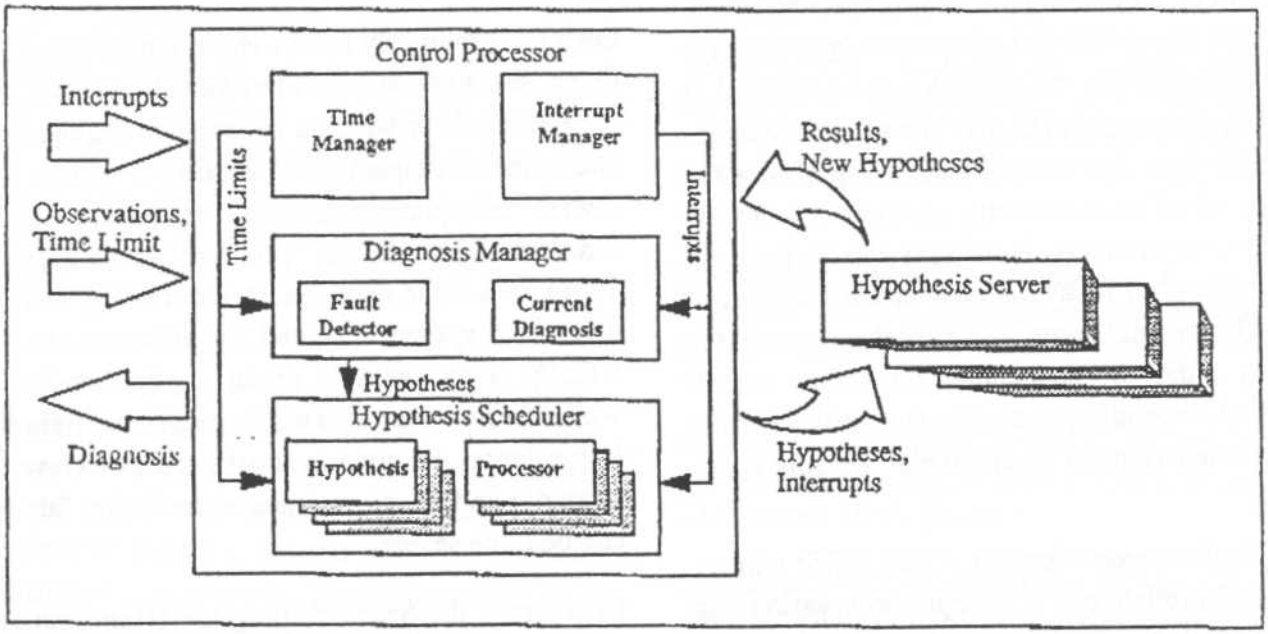

Figure 3. The QRSt Architecture. The Time Manager ensures that time constraints on processing are met. The Interrupt Manager allows the diagnosis to be interrupted when other, higher priority, processing requirements occur.

5.1 Time–Constrained Diagnosis

The basic time–constrained QRS architecture requires only a few modifications to pQRS. First, pQRS is modified to accept a time limit as one of its inputs, along with the set of observations. This time limit is the amount of time that qualitative reasoning may execute on a set of observations before a diagnosis is required.

Second, a Timer Process is added to the Control Processor. The Timer Process is essentially an alarm clock that is set to the allotted time when observations arrive. When the time limit is reached, the alarm clock goes off.

Both the Diagnosis Manager and Hypothesis Scheduler listen to the alarm clock. When the time limit is reached, the Diagnosis Manager returns the most detailed diagnosis that is currently available.

The Hypothesis Scheduler responds to the alarm by reclaiming processing resources. The first action it takes is to dequeue all pending hypotheses for that set of observations. Second, it issues interrupts to each of the active Hypothesis Servers that are evaluating hypotheses for this set of observations. As each Hypothesis Server terminates, it informs the Hypothesis Scheduler of its availability and is returned to the Processor Queue. The hypothesis processes may either be suspended for future use or killed, depending on storage constraints and the expected value of saving the processing state.

With these extensions to pQRS, QRSt will respond predictably. It accepts observation and returns its best assessment of the situation within the prescribed time limit.

5.2 Interruptible Diagnosis

In addition to supporting time–constrained inferencing, QRSt must also be interruptible. Qualitative reasoning may be just one of a number of resources available in an overall status monitoring/diagnostic system. Thus, QRS may be in competition with other software modules for computing resources. Situations may arise in which resources allocated to QRS need to be applied to other more urgent processing needs. By making QRSt interruptible, it can stop work on a problem when another, higher priority, processing requirement occurs.

To support interruptibility, an Interrupt Manager needs to be added to the QRSt architecture. The Interrupt Manager receives interrupts and relays them to the Diagnosis Manager and Hypothesis Scheduler. These modules respond to the interrupts in the same manner as to timer interrupts.

6. pQRSt Testbed

QRS has previously been used in a number of diagnostic applications. They vary in terms of the time scale at which a response is required and in the consequences of missing deadlines. Two major application areas illustrate these differences. The HELIX program (Hamilton 1988) is a helicopter status monitoring and diagnostic system designed for airborne use. HELIX is an example of an application in which the time frame for response is rather tight (typical response times are on the order of a few seconds) and the consequences for being late can be quite severe.

In contrast, the Sherlock program (Hamilton, 1987) is a ground–based gas turbine engine diagnostic system. Timing constraints in Sher-

lock are far less stringent, both in terms of the required response time and the consequences of failing to meet the deadline. In both cases, though, there is a requirement to produce the best assessment within some reasonable time frame.

QRS is written in Common Lisp and has been hosted on a variety of hardware platforms (Symbolics, Sun, Compaq). The development environment for pQRSt is a network of Symbolics workstations (one XL400 and five 3640s). Both the HELIX and Sherlock applications will serve as test applications for evaluating the system.

7. Discussion

The proposed architecture for pQRSt satisfies a key requirement for real–time computing that is often missing in real–time AI applications: predictability. By supporting time–constrained and interruptible operation, this architecture makes the QRS hierarchical diagnostic process suitable for use in real–time status monitoring and diagnostic systems. In addition, the extensions designed to support parallel evaluation of hypotheses should provide both improved performance and greater flexibility in controlling search.

Implementation of the pQRSt architecture has been initiated. Experiments with the system should help to answer questions about both the number and configuration of processors required to achieve desired response times for different applications. Further research must build on related work on operating systems to support real–time AI (Stankovic *et. al.* 1989) and on planning the problem–solving process using approximate reasoning techniques (Lesser *et. al.* 1988).

8. References

Abbott, K. H: Robust Operative Diagnosis as Problem Solving in a Hypothesis Space. *Seventh National Conference on Artificial Intelligence*, St. Paul, Minnesota, 1988.

Davis, R.: Diagnostic Reasoning Based on Structure and Behavior. *Artificial Intelligence*, 24, 1984, 347–410.

de Kleer, J. and Williams, B.C.: Diagnosing Multiple Faults. *Artificial Intelligence*, 32, 1987, 97–130.

Doyle, R. J., Sellers, S. M., and Atkinson, D. J.: A Focused, Context–Sensitive Approach to Monitoring. *Eleventh International Joint Conference on Artificial Intelligence*, Detroit, Michigan, August 20–25, 1989, 1231–1237.

Hamilton, T. P.: HELIX: An Application of Qualitative Physics to Diagnostics in Advanced Helicopters. *International Journal for AI in Engineering*, Vol. 3, No. 3, pp 141–150, July, 1988.

Hamilton, T.P.: Applications of Qualitative Reasoning to Aircraft Diagnosis and Maintenance. *Artificial Intelligence and Robotics Symposium*, June 16–17, 1987, Norfolk, Virginia.

Lesser, V. R., Pavlin, P, and Durfee, E.: Approximate Processing in Real–Time Problem Solving. *AI Magazine*, Vol. 9, No. 1, Spring, 1988, 49–61.

Stankovic, J. A.: Misconceptions About Real–Time Computing. *IEEE Computer*, Vol. 21, No. 10, October, 1988, 10–18.

Stankovic, J. A., Ramamritham, K., and Niehaus, D.: On Using the Spring Kernel to Support Real–Time AI Applications, *Proc. Euro-Micro Workshop on Real–Time Systems*, June 1989.