AN EXPERT SYSTEMS APPROACH TO REALTIME, ACTIVE MANAGEMENT OF A TARGET RESOURCE

David A. Klein

MS-CIS-85-40

Department Of Computer and Information Science
Moore School
University of Pennsylvania
Philadelphia, PA 19104

September 1985

Acknowledgements: This research was supported in part by DARPA grants NOOO14-85-K-0018 and NOOO14-85-K-0807, NSF grants DCR-86-07156, DCR8501482, MCS8219196-CER, MCS-82-07294, 1 RO1-HL-29985-01, U.S. Army grants DAA6-29-84-K-0061, DAAB07-84-K-F077, U.S. Air Force grant 82-NM-299, AI Center grants NSF-MCS-83-05221, U.S. Army Research office grant ARO-DAA29-84-9-0027, Lord Corporation, RCA and Digital Equipment Corporation.

ABSTRACT

The application of experts systems techniques to process control domains represents a potential approach to managing the increasing complexity and dynamics which characterizes many process control environments. This thesis reports on one such application in a complex, multi-agent environment, with an eye toward generalization to other process control domains.

The application concerns the automation of large computing systems operation. The requirement for high availability, high performance, computing systems has created a demand for fast, consistent, expert quality response to operational problems, and effective, flexible automation of computer operations would satisfy this demand while improving the productivity of operations. However, like many process control environments, the computer operations environment is characterized by high complexity and frequent change, rendering it difficult to automate operations in traditional procedural software. These are among the characteristics which motivate an expert systems approach to automation.

JESQ, the focus of this thesis, is a realtime expert system which continuously monitors the level of operating system queue space in a large computing system and takes corrective action as queue space diminishes. JESQ is one of several expert systems which comprise a system called Yorktown Expert System/MVS Manager (YES/MVS). YES/MVS automates many tasks in the domain of computer operations, and is among the first expert systems designed for continuous execution in realtime. The expert system is currently running at the IBM Thomas J. Watson Research Center, and has received a favorable response from operations staff.

The thesis concentrates on several related issues. The requirements which distinguish continuous realtime expert systems that exert active control over their environments from more conventional session-oriented expert systems are identified, and strategies for meeting these requirements are described. An alternative methodology for managing large computing installations is presented. The problems of developing and testing a realtime expert system in an industrial environment are described.

TABLE OF CONTENTS

1. INTRODUCTION

The application of experts systems techniques to process control domains represents a potential approach to managing the increasing complexity and dynamics which characterizes many process control environments. Expert systems may be of potential value as high-level supervisors in current computer-controlled systems which employ rigid, predetermined control sequences [1] (e.g., automated manufacturing systems), and in domains which currently require human operators to monitor and control some process on a continuous basis (e.g., power generation plants, refineries). This thesis reports on the application of expert systems techniques in a continuous, realtime, process control domain where currently both computers and human operators represent active agents which impact the environment: the domain of computer operations.

The requirement for high availability, high performance, computing systems has created a demand for fast, consistent, expert quality response to operational problems. Effective, flexible automation of computer operations would satisfy this demand while improving the productivity of operations. However, like many process control environments, the operations environment is characterized by high complexity and frequent change, rendering it difficult to automate operations in traditional procedural software. Yorktown Expert System/MVS Manager (YES/MVS) offers a preliminary solution through the use of expert systems techniques, providing a basis for automation which is powerful enough to accommodate the complexity of large system operation, yet flexible enough to endure frequent modifications as requirements change.

YES/MVS is an experimental expert system which assists in the realtime operation of a large IBM Multiple Virtual Storage (MVS) Job Entry Subsystem III (JES3) system (henceforth referred to as the *target system*). YES/MVS emulates a human operator, receiving messages which would normally appear at an operator's console, submitting queries and analyzing replies to ascertain the status of the target system, and submitting active commands to MVS and its subsystems to perform operational tasks.

YES/MVS addresses both routine actions taken in operating the target system and spontaneous problems which would normally be handled by an operator. These include various MVS-detected hardware errors, the depletion of operating system queue space, problems of transmission between computers installed at the same site, abnormal termination of subsystems, and others. YES/MVS is organized as several logically distinct expert systems, each encoding the expertise required to perform operational tasks in one of these problem domains. In some sense, this decomposition of the operations management domain into discrete problem domains is artificial, motivated by the circumscription of perspective required to produce high-performance expert systems, by effective project organization, and by the goal of modular knowledge encoding. For example, hardware errors, channel-to-channel link problems, and output processing problems all impact JES queue space, but these are addressed by separate expert systems in YES/MVS.

The expert system that deals mostly with output processing bottlenecks and associated devices as they impact JES queue space is called JESQ, and is the work of the author. JESQ continuously monitors the status of operating system queue space, taking corrective action as queue space decreases. Operations management is concerned with monitoring the remaining available queue space because its depletion requires restarting the system, potentially inconveniencing all system users for a substantial period of time. JESQ runs regularly at the IBM Yorktown Computing Center and has received a favorable response from the operations staff. The system runs in advisory mode (gives advice for operator review) during the day, and fully authorized (submitting active commands to the target system) at night. JESQ is of interest from the perspectives of both management and computer science. In particular, this thesis provides the following contributions:

- An alternative methodology for managing large computing installations is suggested. Employing a computer program as the primary executing agent of operational policy should create an environment characterized by timeliness of policy maintenance, fast and consistent execution of policy, knowledge-intensive operational decision-making, data-intensive operational decision-making, decreased labor requirements, greater installation-wide awareness of policy, decreased

dependence on specific personnel, and ease of policy testing and enhancement. Such an environment would facilitate the achievement of computing installation's goals, increasing the availability and performance of existing computing resources, promoting the productivity of operations staff, reducing labor costs, and providing a partial solution to the problem of high turnover in operations personnel. While this work concentrates on the domain of computer operations, similar benefits may potentially be realized in other process control domains through the employment of expert systems techniques.

- General issues that arise in the construction of active, realtime expert systems (as contrasted with more traditional session-oriented expert systems) are identified, and strategies and mechanisms for dealing with some of these in the queue space management domain are described. JESQ and other YES/MVS domain expert systems are among the first which operate continuously in realtime and exert active control over the environments they monitor. Many of the methods employed in JESQ which accomplish realtime, active control may potentially be applicable in other process control expert systems domains. An abstract problem description is included in the thesis to facilitate the identification of domains which are similar to the domain of queue space management.

- One fully-implemented architecture which integrates decision-making and acting in a realtime, multi-agent environment that is sufficiently complex to discourage explicit modelling of all interactions is described. An alternative architecture is outlined and contrasted with the existing implementation.

- An account of the methodology under which JESQ was developed is provided, supplementing the relatively sparse case data on the construction of expert systems in industrial environments. Of particular interest, the difficulties of testing an expert system in a complex, realtime environment are illuminated.

The thesis is organized as follows. Section 2 examines the domain from several perspectives. A general description of problems in computer operations management illuminates the motivation for automating operations. Subsequent sections describe the difficulty of automation in a dynamic environment where problem resolution relies on complex, loosely structured heuristic strategies, motivating an expert systems approach. A review of realtime expert systems follows, creating a context in which to consider JESQ. Next, the JES queue space management domain is described in detail, followed by a more abstract description that provides a basis for comparison with other domains. Section 2 concludes with a discussion that contrasts the implementation requirements and strategies of JESQ with those of more conventional consultative expert systems. Section 3 describes the architecture of JESQ. The first subsection justifies the level of user modelling implemented in JESQ in the context of the operations management hierarchy. Following is a simplified review of the production systems architecture, the skeletal framework in which JESQ is implemented. Next, an overview of the organization of JESQ's knowledge base is presented, followed by a description of the key mechanisms implemented and their underlying design motivations. With the needed background provided, the following section depicts a detailed execution trace of JESQ resolving a JES queue space problem. Finally, JESQ's position within the YES/MVS architecture and operations environment is described. Section 4 identifies some of the challenges of developing JESQ, including a discussion of knowledge acquisition, testing and knowledge base refinement, and requirements for formal evaluation. Section 5 is a critical review of JESQ, describing an alternative architecture and directions for further research. Section 6 summarizes the ideas presented. Section 7 contains references.

2. PROBLEM CHARACTERIZATION AND APPROACH

This section introduces the focal issues of this thesis from several perspectives. Section 2.1 outlines the motivation for this work, illuminating the difficulties of managing large computing installations by standard methods. Section 2.2 suggests automating operations as a solution to many of these difficulties while identifying the problems of using traditional programming techniques in an environment characterized by complexity, ad-hoc methods, and frequent change. Section 2.3 argues that employing expert systems technology renders automation feasible in such an environment. Section 2.4 provides an overview of the few expert systems that have been developed for continuous realtime operation, creating a context in which to consider the focal operations problem domain of this thesis -- JES queue space management -- in section 2.5. Section 2.6 gives an abstract characterization of the queue space management domain, providing a basis for identifying similar problem domains. Section 2.7 contrasts the implementation requirements for automation in this active, realtime domain with those of more traditional session-oriented expert systems, and outlines associated implementation strategies. A summary of the ideas presented appears in section 2.8.

2.1 PROBLEMS IN COMPUTER OPERATIONS MANAGEMENT

Large computer installations are continuously monitored by a team of computer operators. Operators concurrently perform many routine tasks including mounting machine-readable tapes, loading printers with paper, answering phones, and actively monitoring the condition of the computer system. Operators additionally watch a number of consoles for a variety of messages that may be volunteered by the computer system, responding to the problems indicated by such messages as they arise. Problem resolution usually involves submitting queries to the computer system in order to ascertain the status of key system resources, consulting voluminous system documentation, and manipulating system parameters via the submission of corrective commands. In some cases, the operator may consult

systems programmers, systems engineers, or other sources of expertise, although operator action is frequently required in a timeframe that prohibits consultation.

While each of the operator's potential actions is conceptually simple and easily performed, his job is nonetheless complex in that a significant breadth of knowledge is required to perform the full range of operator responsibilities. Operations managers typically develop standards and procedures to reduce this complexity, providing preconceived responses to anticipated problems and "cook book recipes" for routine operations. The goal of such standardization is to allow the installation to be managed as a typical production process, where data is the raw material input to the process and satisfactory system services represent the commodity produced.

In practice, this management philosophy is difficult to implement because the operations environment is so dynamic. With hardware and software evolving rapidly, new components are frequently introduced into the operations environment. Modifications to existing components are also common. Even installations with relatively static hardware and software configurations are nonetheless faced with managing temporarily altered environments when key subsystems experience problems or workload deviates significantly from the norm. Naturally, the dynamic nature of the environment renders it difficult to keep operational procedures current.

Where operational procedures lag behind the environments to which they pertain, operators experiment and develop ad-hoc solutions to system management problems. Particularly effective solutions are discussed among the operators and become part of the installation's "operational folklore". Unlike formal procedures, these operational rules of thumb are not documented and distributed, and thus, are not performed consistently by all operators. Consequently, they cannot be subjected to periodic management review, nor are they immediately intelligible to systems programmers who later examine the effects of operator actions in addressing system problems. Of course, this periodic migration from standardization to "folklorization" does not impact the installation immediately. Rather, management slowly loses control of operations. Until this becomes apparent, ad-hoc problem resolution may actually be encouraged by management in favor of reviewing and updating operational

policy. The result is a somewhat personal approach to problem solving that varies from operator to operator and increases the installation's dependance on specific personnel.

While standardization is necessary in any complex production environment, high turnover in operations staff makes such management controls particularly important. A long training period is required to produce a skilled operator at a given installation. Even the training of newly-hired experienced operators can represent a significant cost to the installation, since operators' responsibilities vary across installations with management policy, with the particular systems installed at a site, and with workload characteristics. The problem of operator turnover is exacerbated by the common management practice of promoting the best operators to systems programmers.

Thus, computer operations is characterized by complexity and dynamics (with regard to both the technical environment and its personnel), rendering it difficult to manage computer installations by standard methods.

The domain of JES queue space management exemplifies this difficulty. JES queue space is a common resource (disk storage) in IBM environments for the staging of computer jobs before, during and after execution. Jobs are normally deleted from the queue space once output has been completed to a printer, a transmission line, or other output medium. Operations management is concerned with monitoring the remaining available queue space because its depletion requires restarting the system, potentially inconveniencing all system users for a substantial period of time.

To maintain queue space when it becomes dangerously low, operators employ a "bag of tricks" which includes, for example, speeding up the printing of completed jobs via the manipulation of system parameters, dumping large print jobs to tape, and refusing new jobs. While queue space management methods might at one time have been uniformly employed by operations staff at the installation where JESQ was developed, personal space management strategies have evolved with new releases of JES, with changing workloads, and with new hardware configurations. That is, each operator executes his or her favorite rules of thumb to free queue space. Since up-to-date procedures no longer exist, the computing center depends on the most experienced operators to handle queue space problems. Be-

cause these operators do not resolve queue space problems uniformly, however, systems programmers cannot readily deduce the cause of a queue space incident: one operator searches the on-line job queue for a huge dataset; another concentrates on expediting output processing. The installation cannot easily review problem situations because there is no longer a standard approach to queue space problem resolution. More importantly, a queue space problem can exist for hours because no single operator holds the entire bag of tricks.

The shortage of skilled operators, the increasing complexity and speed requirements of the operator's job, and the dynamic nature of the installation call for more powerful operations management tools. In particular, tools are needed to ease the workload of the operator, to provide fast, consistent reactions to installation problems, to decrease the installation's dependence on specific personnel, to provide a basis for enforcing installation management policy, and to provide for the orderly integration of new policy with old.

2.2 AUTOMATION OF COMPUTER OPERATIONS

Many computer operations management problems could be eliminated by automating operational procedures in software. In particular, automated operations would offer the following advantages over manual operations:

- *Imposed Timeliness of Policy Maintenance:* Because the program would serve as the primary executing agent of operational policy, management would be encouraged to keep it up to date. Allowing policy to lag behind the environment would no longer represent a feasible strategy, since policy would be directly executed and the role of human operators as ad-hoc problem solvers would be diminished. The notion that policy need not be deployed to all operators immediately with each modification would also encourage timely policy maintenance, in that implementation of new policy would be simplified. That is, the processes of "run book" (a book of operational standards and procedures) revision and operator retraining would be reduced to

a modification in the automated operator program. Thus, the introduction of the automated operator program would alter the structure of work in the operations environment such that timely maintenance of operational policy would be encouraged.

- *Speed:* Actions not requiring manual intervention would be executed at machine speeds rather than at human speeds.

- *Knowledge-intensive Decision-making:* Ideally, the computer program would represent an installation-wide fusion of operations knowledge. Thus, the program could potentially contain more knowledge than any single individual at that installation.

- *Data-intensive Decision-making:* Decisions could be based upon volumes of system data not easily organized or comprehended by a human operator.

- *Tirelessness:* An automated operator would perform repetitive tasks without taking breaks.

- *Decreased Manpower Requirements:* Human operators would be needed only to handle exceptional incidents, to perform manual tasks, and to monitor the automated operations program.

- *A Tool for Operator Training:* Augmented with friendly interfaces, the automated operator could be used as a training tool to keep the remaining (human) operators and other installation personnel apprised of current policy.

- *Consistency:* Reactions to incidents would be consistent, and in precise accordance with adopted policy. This would facilitate the review and development of policy by both management (the designers of policy) and troubleshooters (the reviewers of operator actions in problem situations).

- *Decreased Dependence on Specific Personnel:* Procedures would be encoded in a program rather than in the "private knowledge stores" of the most experienced operators.

- *Policy Testing and Enhancement:* Policies recorded on paper for reference by human operators are subject to individual interpretation. In contrast, a program which encodes such procedures would be applied directly (without an interpretive component in between) so that the effects of new policy could be better isolated for analysis.

The potential advantages are obvious, and established techniques of control theory have already been applied to similar problem domains that are highly constrained and amenable to analytic treatment [1]. However, the domain of operations (the queue space management domain in particular) is not well formalized and is sufficiently complex to defy single-pass development of an automated system. Queue space maintenance, for example, is accomplished via loosely structured heuristic methods rather than highly formalized techniques, and these are subject to frequent change as the installation evolves.

These domain characteristics would render it difficult to implement and maintain an automated operations program written in a conventional procedural programming language. Procedural formalisms would require that the programmer anticipate and write code to handle all permutations and combinations of potential operational events, an impractical task given the complexity of the operations environment. Since program control constructs are tightly coupled with the domain knowledge encoded in procedural programs, maintaining a procedural automated operator program would be difficult. In addition, review of operational policy (the program) would be impractical without the aid of a programmer under the procedural framework. These are among the reasons why such knowledge should not be embedded in the operating system itself.

What is required, then, is a tool for automating operations that can accommodate the complexity, dynamics, and lack of formalization which characterizes the operations environment. In particular, the tool must provide for:

- *Incremental development:* Because the knowledge to be captured is complex and difficult to organize and communicate, the automated operator must be developed incrementally. The complexity and breadth of the knowledge that must be encoded prohibits the completion of the

automated operator in a single iteration, requiring the capability to incrementally add and modify operational policies.

- *Ease of ongoing maintenance:* Since installation policy will change frequently to reflect environmental modifications, the need for ongoing maintenance requires minimal dependence between the chunks of knowledge encoded in the program. Ideally, we want to be able to remove and add management policies as units. That is, to the extent possible, a management policy should map directly into a set of isolated programming constructs. The collection of such constructs would constitute the installation's operational policy. The installation's operational policy, then, would reside in one location, recorded in a uniform format which facilitates ongoing maintenance.

- *Understandability:* Installation policy must be accessible to those involved in running the installation. Thus, the programming constructs that represent policy must be understandable to operations personnel.

Some expert systems formalisms -- *production systems* in particular -- encompass these characteristics.

2.3 AN EXPERT SYSTEMS APPROACH

Expert systems or *knowledge-based systems* are a product of research in the field of artificial intelligence which have recently demonstrated value in solving practical problems [2]. This relatively new software technology has already been applied to a wide variety of useful (but still experimental) applications. These include fields as diverse as medical diagnosis [3], mineral exploration [4], and chemical analysis [5].

"Broadly speaking, an expert system is a computer program that uses explicitly represented knowledge and computational inference procedures to solve problems normally requiring significant human expertise" [6]. The development of an expert system involves extracting knowledge of a problem domain from human experts and encoding their expertise in a *knowledge base.* In the production

systems formalism, the knowledge base is a declarative structure composed of IF ... THEN rules, where the antecedent (IF part) of a rule describes the conditions under which the actions in its consequent (THEN part) should be executed. In JESQ, the antecedents refer to the state of the system being controlled and the consequents list the appropriate actions corresponding to that state. (This is an oversimplified description of the system architecture which is further discussed in section 3).

For example, Figure 1 depicts the English equivalent of a policy rule in JESQ's knowledge base.

```
IF  remaining JES queue space is low;
and there is a 3211 printer in the configuration;
and that 3211 is active;
and that 3211 is set to print only small jobs;
and there are few small jobs to print (the printer is about to idle);
and there are large jobs waiting to print;
THEN
    submit to the target system that command which increases
      the 3211's line limit, allowing the large jobs to print,
        thereby freeing queue space.
```

Figure 1: JESQ Policy Rule (English Equivalent)

Since no rule explicitly "calls" another [7], modularity of knowledge encoding is encouraged, allowing the addition and deletion of operational policies (rules) with minimal global computational impact as compared with the equivalent procedural program. This quality of expert systems facilitates incremental development, ongoing modification, and understandability. Mapping each statement of policy into a set of rules where possible, policies can be added and deleted more easily than in procedural software as the installation evolves.

Of course, some organizational changes would be required to accommodate an expert systems approach to operations management. In particular, a knowledge base maintenance function would replace that of run book maintenance. Ideally, one person (perhaps a systems programmer) would coordinate the modification and testing of the knowledge base to reflect new and modified management policies, creating a control point for integrating emerging operational folklore with existing policy. While the cost of maintaining the knowledge base might exceed that of the run book (the knowledge base is undoubtedly less flexible than its paper counterpart), achieving the improvements in management control, system reliability and availability, and installation staff productivity potentially provided by automating operations with expert systems would more than offset this cost.

2.4 REVIEW OF REALTIME EXPERT SYSTEMS

Most expert systems applications to date are session-oriented, assuming a static external world for the duration of a session. Only a few expert systems have been developed in domains where realtime continuous execution is required, and these do not generally exert any active control over their environments. Rather, such systems alert human operators to potential or actual problem situations in the environments they monitor. Most of the systems have only been tested in simulated environments.

Some examples are discussed below. The final system discussed exerts active control over its environment, but its authors have not labelled it an expert system.

Realtime Sensor-Based Diagnosis of Machine Faults

PDS [8] performs realtime detection and diagnosis of malfunctions in machine processes driven by data received from sensors. PDS is a forward-chaining (i.e., data driven forward reasoning rather than goal driven backward reasoning) rule-based system implemented in SRL [9]. The system alerts human operators (e.g., triggers an alarm) when a problem is detected. The novelty of PDS lies in its ability to reason about and deal with spurious sensor readings and sensor degradation, implementing techniques which Fox calls *retrospective analysis* and *meta-diagnosis* as solutions to these problems.

Realtime Air Traffic Control Simulation

Wesson's work [10] combines planning techniques with discrete simulation to perform the task of the air traffic controller. The purpose of the system is to create in realtime a minimal sequence of aircraft commands which guarantees a specified time period free from aircraft conflicts. Decisions are based on knowledge of the current state of the world and aircraft intentions, simulating forward in time to

observe how expected results conform to a global strategy. Planned actions are generated by event-response pairs that are reminiscent of the production systems formalism, although these are embedded in procedural (PASCAL) code "for efficiency and generality". The system has been tested with simulated test cases.

Realtime Diagnosis and Treatment of Nuclear Reactor Accidents

REACTOR [11] is an expert system which assists operators in the diagnosis and treatment of nuclear reactor accidents. The system monitors a nuclear reactor facility, detects deviations from normal operating conditions, determines the significance of the situation, and recommends an appropriate response.

REACTOR's knowledge base contains *function-oriented knowledge* and *event-oriented knowledge*. Function-oriented knowledge concerns the configuration of the reactor system and the relationships between its components in performing a given function. Event-oriented knowledge describes the expected behavior of the reactor under known accident conditions in the form of production rules. The system reasons forward from known facts until a conclusion can be reached. If not enough information is available to reach a conclusion, the system reasons backward to determine what information it needs to know and then queries plant instruments to fill the gaps in its knowledge. If an accident cannot be diagnosed using the event-oriented approach, the function-oriented strategy is used, employing a tree of paths which can be used to provide a given safety function (the *response tree*).

REACTOR is implemented in LISP and FORTRAN, and has been tested with a reactor simulator. REACTOR advises a human operator, but maintains no active control over the reactor facility.

Realtime Alarm Analysis in Chemical Process Plants

FALCON [12] identifies the probable causes of disturbances in chemical process plants, employing knowledge of the immediate effects induced by a fault in a given component and knowledge of how disturbances in the inputs of a normal component will propagate to disturbances in the outputs. The monitor module in FALCON tests sensor data against prespecified ranges of acceptability to identify a disturbance. Upon detection of a disturbance, the fault analyzer module reasons about probable causes.

Two versions of the fault analyzer have been developed. The first version reasons directly from observed data, employing a forward-chaining rule-based formalism implemented in LISP. The second version examines an explicit model of the process to find possible causes of a disturbance, reporting and explaining those which are most likely. This model is represented in a network of: local disturbances caused by faults in individual components, propagation descriptions, and exceptions to the normal diagnosis process.

FALCON informs human operators of detected faults and answers their queries about its analysis, but takes no corrective action on its own. It has been tested against a simulator.

Realtime Ventilator Management in the Intensive Care Unit

VM [13] interprets on-line physiological data used to manage post-surgical patients receiving mechanical breathing assistance in the intensive care unit. The system detects possible measurement errors, recognizes untoward events in the patient/machine system and recommends corrective action, summarizes the patient's physiological status, suggests adjustments to the patient's therapy, and maintains a set of patient-specific expectations over time. A set of production rules encodes clinical interpretation knowledge obtained from medical experts. A forward-chaining version of the MYCIN interpreter [3] is employed to accommodate the data-driven problem domain. Data is obtained from patient monitoring sensors in the intensive care unit.

Realtime Control of a Manufacturing Cell

Work at Carnegie-Mellon University's Robotics Institute and Westinghouse Corporation [14] involves the construction of a flexible manufacturing cell for open-die forging (swaging). Data is provided to the supervisory system by swaging cell sensors, although questionable readings are not scrutinized by the system.

A forward-chaining production systems architecture drives the movement of robots which perform the swaging task. The antecedents of rules describe the conditions under which a robot action is appropriate and the consequents give a procedure for performing that action. Processing errors are accounted for to some extent in that a given action will not be executed unless the required preconditions have been satisfied. However, inspection and diagnosis of the actual parts produced is performed in a post-processing mode (i.e., not in realtime). The eventual goal of this work is "to instrument the machines so that the operations can be carried out in a fully unmanned sequence while still correcting for errors".

Summary

A handful of expert systems which operate continuously in realtime have been developed in a diverse set of domains. In general, these systems monitor processes in their environments and alert a human operator to potential and actual problems. The approaches employed include encoding event-response pairs, simulation under explicit models of environmental processes, planning based upon state variables and environmental agent intentions, and combinations of these. In general, these expert systems do not exert active control over resources and have been tested in simulated environments.

With this brief review of realtime expert systems provided, the following section describes the domain of primary interest in this thesis.

2.5 DOMAIN OVERVIEW: JES QUEUE SPACE MANAGEMENT

JES queue space is a common resource (disk storage) in IBM system environments for the staging of computer jobs before, during and after execution. Jobs are normally deleted from the queue space once output has been completed to a printer, a transmission line, or other output medium. JES queue space is also used by JES itself as a scratch area for executing its functions. In addition, JES maintains batch job output for online viewing (via IBM's Time Sharing Option (TSO) software) in the JES queue space area.

Operations management is concerned with monitoring the remaining available queue space because its depletion requires restarting the system, potentially inconveniencing all system users for a substantial period of time. Of course, the problem could be eliminated by employing the "brute force" strategy of allocating more and more disk storage to JES as needed (although this allocation is fixed at system startup time). However, this trade-off of effective space management for additional physical storage is generally regarded as suboptimal, and represents an expensive temporary "fix" in the absence of identifiable increases in system workload.

The operator may take several protective and corrective actions when queue space begins to diminish, and these may be described in terms of three general goals:

- *Protect Remaining Queue Space*: The operator must protect the space that remains when dangerously low (e.g., 5%). For example, the operator may vary the main processor offline, blocking the initiation of additional jobs which could generate output on the queue.

- *Free Queue Space*: The operator can manipulate various devices and operating system parameters to free queue space. For example, the operator may run **DJ** (for Dump Job) to copy large jobs from the queue to tape. Jobs are reinstated on the queue when space improves following

conversations with their owners, provided that the jobs are truly needed. Alternatively, the operator may change parameter settings on printers to allow jobs with special characteristics (e.g., special paper or security requirements) to print. The operator may also change the maximum line count limits on printers set to favor small jobs in cases where large jobs are waiting and small jobs will soon all be printed. The operator can additionally reroute large jobs destined for slow printers to faster printers with a relatively light load.

Since space-freeing actions may be time-critical when queue space is dangerously low, the operator may allocate resources in preparation for these actions well in advance of their execution, often before their specific use is identified. For example, DJ requires that a tape be mounted and the DJ job itself be initiated before any jobs can be copied to tape using this facility. Similarly, the printing of special forms often (depending on current printer settings) requires a microcode change (floppy disk insertion) on the printer and a paper change. Ideally, the operator performs such set-up actions as soon as a potential incident is identified, even though they may be in vain. Of course, the environment must be restored to its original state once a queue space problem has been resolved.

- *Diagnose and Eliminate the Cause(s) of Queue Space Depletion*: In some cases, there exists a direct cause-effect relationship between the actions of an environmental agent (e.g., user, operator, device) and a queue space problem. For example, a printer might not be operational, or a user may be storing a million-line dump on the queue. In such cases, the operator must correct the problem as well as restore the queue to an acceptable state in a reasonable amount of time.

While this description imposes some order on the space management process, again, there is no standard methodology for managing JES queue space. Rather, each operator performs some subset of these actions according to his or her breadth of knowledge and style. Note that some judgement is required on the part of the operator in choosing among competing actions. For example, output stored for online (TSO) viewing can be purged from the queue by using DJ, by requesting action from

the user himself, by printing the job, or even by deleting the job. The techniques employed are inherently heuristic in nature.

2.6 ABSTRACT DOMAIN CHARACTERIZATION

The need to continuously monitor and actively maintain the level of a critical resource in real time arises in several complex real-world domains. Obvious examples include controlling temperature in an enclosed area, maintaining the electrical output of a power plant, and controlling the humidity level in a greenhouse.

It therefore is useful to generalize from the description in section 2.5 so that domains which are similar to the space management domain can be more readily identified and some of the techniques employed in building JESQ can be implemented in similar systems.

The JES queue space domain encompasses the following abstract characteristics:

- *There exists a target resource which need be continuously monitored and maintained.* For example, JES queue space is monitored periodically by a human operator.

- *There exists an identifiable goal state for the target resource.* That is, a notion of an acceptable level of the target resource can be formulated. For example, the goal state for the operating system queue is 25% or more space left.

- *There exists a gauge by which the state of the target resource can be repeatedly measured.* In the queue space domain, the gauge is a response to an operating system query which gives the amount of space left on the queue, expressed as a percentage.

- *The state of the target resource is a function of the behavior of several active agents (humans and resources) in the environment.* For example, excessive output generated by executing jobs, offline printers, and inappropriate parameter settings on output devices all contribute to queue space depletion. Operator actions, cooperative user actions, and normal processing of output all con-

tribute to queue space renewal. Once installed, the expert system itself is an important agent in this multi-agent environment.

- *The target resource can be restored to the goal state via the application of abstract operators.* This is just to state that the relationship between the actions of agents and the state of the target resource is goal-oriented rather than random. Agents can execute plans to restore the target resource to its goal state. Agents can cooperatively execute subplans to achieve the goal state.

- *The states of some environmental agents can be ascertained from sensors upon request, while the states of others cannot.* For example, the names of jobs queued for a particular printer and the status of the main processor under the operating system can be derived from responses to operating system queries. Printer paper jams, the operator's availability to file confidential output, and a user's plans for a large dataset in held status are examples of environmental agent states which cannot be ascertained directly from sensors.

- *The environment is sufficiently complex to discourage explicit modelling of all interactions.* The complexity of a multiple-CPU operations environment running hundreds of system software modules under the direction of hundreds of batch and online users and several human operators prohibits explicit modelling of all interactions between resources that may affect JES queue space. This characteristic prohibits diagnostic approaches such as those used in REACTOR and FALCON.

- *Actions must sometimes be performed in a timeframe that prohibits the satisfaction of the associated preconditions.* Some actions have preconditions, requiring that setup actions be performed (e.g., mounting a tape for DJ). In some cases, an action must be performed immediately upon recognition of its triggering context, allowing virtually no time to satisfy the preconditions for that action (e.g., starting DJ with 1% space left on the JES queue).

- *There is an unpredictable temporal delay between the initiation of an action and its execution.* For example, a command submitted to the operating system may be enqueued (depending on mes-

sage traffic), paging delays might occur, or commands of higher priority may be submitted by other operators. In the degenerate case, a command may be lost in transmission so that the "delay" is infinite.

- *The effectiveness of actions cannot be precisely computed at the time of their submission.* For example, while raising the line count limit on a printer will allow larger jobs to print, the eventual effect on JES queue space is not entirely clear. Even though the line counts of jobs queued for that printer are known at the instant the line limit is raised, new jobs can complete and be printed on that printer at any moment. Jobs queued for other printers could be routed to the manipulated printer as well. In the negative case, the command which raises the line limit could fail, producing no result at all.

- *Potential actions may compete for resources.* For example, the operator may choose between printing special forms on a particular printer and routing a normal-form job to that same printer.

- *Some actions are performed over a temporal interval, while others are initiated at a point in time.* For example, the operator may notify a user to take action on a held job and subsequently transfer that job to tape over some temporal interval. In contrast, the operator may initiate the rerouting of a large job to a print on a faster printer at a point in time.

- *Operation of the environment is characterized by competing qualitative objectives.* For example, cancelling large jobs will free queue space but this action will anger the affected users. Alternatively, allowing queue space to remain at a low level will degrade performance, angering the remaining users. Dumping the large jobs to tape will tie up a tape drive, possibly causing executing jobs to wait for a free drive. Also, dumping a job to tape is more labor-intensive than cancelling the job, and the operator may desire the quickest solution so that he can attend to other space-freeing tasks.

In summary, the environment is characterized by competing objectives and actions in a multi-agent environment which is sufficiently complex to discourage explicit modelling of all interactions. An

expert system in such a domain must be robust, accommodating unexpected state changes at any time and lost or unreliable data. The following section considers these characteristics in the context of implementation requirements for JESQ.

2.7 IMPLEMENTATION REQUIREMENTS AND STRATEGIES

The construction of an expert system which operates continuously in realtime and exerts active control in a multi-agent environment encompasses requirements not normally addressed in session-oriented consultation systems. The following sections identify some of the issues that distinguish JESQ from session-oriented systems, outlining strategies for accommodating some of these differences.

Approximating a Timely and Consistent Incomplete Model of the World

In many session-oriented expert systems, primitive facts about the world are assumed true until explicitly negated. A richer scheme is required for maintaining a model of the world in an expert system which actively solicits frequently changing information in a realtime multi-agent environment. Primitive assertions may be rendered inconsistent with the real world at any time by the actions of environmental agents both known and unknown to JESQ, and by JESQ itself. Periodic reassertion of primitive facts is required in both cases:

1. JESQ cannot know of changes in the world effected by other environmental agents immediately. Assuming that we cannot require all agents to notify the expert system when an action is taken, (i.e., values for state variables are not volunteered by all relevant environmental components as in many process control applications) it is necessary to query the state of world periodically in order to detect such changes.

2. One might assume that the expert system could at least reflect the effects of its own actions in its internal model of the world. However, periodic reassertion of facts about portions of the world affected by expert system actions is nonetheless required here as well: (1) The action may fail due to the state of some variable that is missing from JESQ's internal model of the world, (2) There may be a significant time lag between the initiation of the action by the expert system and the effect of that action on the world, and (3) The effects of the action's successful execution cannot be precisely computed in advance, since state variables may have changed since the time of action initiation. Thus, JESQ does not update its internal model of the environment to reflect the anticipated effects of its actions until such effects are verified by operating system responses. JESQ does, however, mark those state variables that are expected to change as a result of its own actions as "unreliable". Unreliable state variables are disqualified from consideration in making operational inferences until fresh values for them are supplied by subsequent operating system responses.

Thus, in both cases it is required that the JESQ actively solicit data from sensors on a periodic basis. Of course, this strategy does not guarantee the reliability of target system state data, since it is possible that this data is outdated by the time it reaches the expert system, regardless of the interval over which such data is requested. For example, JESQ may receive data regarding the status of a printer a microsecond before its status is changed. JESQ therefore must make decisions based on an approximation of the state of the world.

The world model maintenance strategy employed in JESQ involves periodically reasserting all primitive facts which are not volunteered by sensors, and simply capturing the few facts that are automatically provided. The interval of query submission varies by device according to operator's estimates of the duration of information reliability. In some cases, the interval is adjusted dynamically during execution. Whenever JESQ submits an action command that is expected to change the status of resources represented in its internal partial model of the environment, JESQ labels those portions of the

model as unreliable. Unreliable state variables are ignored in the decision making process until they are reassigned values provided by responses to JESQ's subsequent queries of those variables.

Note that approximating a timely model of the world is more difficult for a human operator than for JESQ. Human limitations render it difficult to keep track of all relevant state variables and to submit queries with JESQ's frequency.

Repeating Failed Paths of Execution

Expert systems that assume a static and closed world can employ backtracking algorithms which prune failed paths of reasoning, disqualifying them from further consideration. In contrast, the strategy in JESQ incorporates the notion that its environment is too complex to model in its entirety, and compensates for the effects of unknown environmental agent actions on relevant state variables by implementing the notion that an unsuccessful attempt to perform an action may succeed on the next try for reasons beyond its knowledge sources. That is, JESQ encodes the notion that actions which fail under a given set of conditions may later succeed under the "same" conditions (as recorded in its internal model of the environment), since the set of conditions considered is assumed to be incomplete.

Knowledge About Temporal Requirements For Execution

The designers of session-oriented expert systems generally need not be concerned about the timeframes in which the advice provided by such systems is formulated. Of course, every system incorporates *some* assumptions of this nature (e.g., consultative systems are restricted to employing inference mechanisms which execute within the temporal limits of a user's patience), but they are not central implementation considerations. In contrast, the design of an expert system that deduces and performs actions in realtime must incorporate approximate knowledge about both its own processing

speed and the speed of agents in its environment. Temporal estimates of concern in designing JESQ included:

- the approximate time required for self-initiated actions to take effect in the real world;
- the approximate time required for the completion of setup tasks which satisfy preconditions for potential expert system actions;
- the approximate time required for human operators and other agents in the environment to complete tasks requiring manual intervention;
- the approximate time required for querying the state of the target resource and other resources in the environment which impact it;
- the approximate time required for collecting responses to queries;
- the approximate time required for reducing primitive assertions to high-level symbolic assertions; and
- the approximate time required for deducing appropriate actions from primitive and high-level assertions.

As discussed in section 2.6, these speeds cannot be precisely computed.

Scheduling the Creation of Primitive Assertions Over Time

Most session-oriented expert systems need not represent the notion of internally scheduled processing over specific temporal intervals. While these systems dynamically schedule the execution of satisfied rules, such scheduling is usually not performed in the context of an explicit representation of time. In contrast, JESQ must schedule its own future events in terms of relative temporal intervals. For example, periodic queries are triggered by the timed creation of primitive assertions (e.g., "it's time for another queue space query"). Another example, JESQ warns users that their datasets may

be removed in 10 minutes in the absence of explicit action on their part, and specifies the creation of a primitive assertion that those 10 minutes have passed so that it can remove such datasets if they still exist at the time this assertion appears in its internal memory.

Satisfying the Preconditions of Goal-Oriented Actions in Advance

"Timeless" systems can employ backward-chaining to determine preconditions for goal-oriented actions or conclusions, instantiating subgoals that satisfy preconditions at the time such principle goals are generated. In contrast, there are situations in the queue space domain that require preconditions for goal-oriented actions to be satisfied well in advance of principle goal generation. This is because some setup actions consume too much time to allow for the successful restoration of the JES queue to its goal state. For example, if a printer must be loaded with special forms to print large jobs at a time when the space remaining on the queue is extremely low (say, 3%), queue space may be exhausted by the time the forms have been loaded. Instead, JESQ would reset the forms at, say 10%, in anticipation of the critical condition, possibly before such special form output has even been generated on the JES queue. Thus, setup actions are performed by JESQ before the target resource reaches a critical level (if possible), even though the associated goals may never be generated.

Monitoring Expert System Support Facilities

Session-oriented expert systems need not reason about or account for the disappearance of the user. Since such systems do not explicitly reason about their own execution time or the timeliness of their facts, they may wait "indefinitely" for input from the user without consequence. In contrast, an active, realtime expert system must monitor not only the resources in its target environment (the application), but also its own support facilities. For example, the system must monitor the status of the target system interface and report its failure to the operator. In the absence of this capability, the

human operator would be unaware, for example, that routine monitoring of JES queue space was not being successfully performed by the expert system.

Actions That Do Nothing

A realtime expert system must have a notion of "doing nothing" or idling, since there will be times when it simply has nothing to do but wait until it is appropriate to reexamine the state of the target resource. Idling in this context is defined to be an action that changes nothing in the real environment or in the internal model of the target system (except for recording the passage of time). This action must be interruptible by the receipt of data from the target system (e.g., a response to a query about an environmental resource) and by an internally scheduled event (e.g., query the target resource in five minutes).

Garbage Collection

The issue of garbage collection can be ignored in session-oriented consultation systems which do not generate enough garbage in a single session to exceed reasonable space allocations. (In this context, the term *garbage* includes tokens which are no longer needed by the expert system (e.g., goals that have already been satisfied) and excludes low level entities (e.g., pointers) that are created and maintained by underlying interpreters which are transparent to the expert system). In contrast, space requirements are infinite for any continuous realtime expert system that does not do garbage collection and generates at least one token on a periodic basis. In particular, the following tokens must be deleted in JESQ:

- primitive assertions that have expired due to age;
- primitive assertions that are assumed unreliable following the execution of expert system actions;

- goals pertaining to actions that have already been performed by another resource in the environment;
- goals pertaining to actions that have already been performed by the expert system itself; and
- conflicting primitive assertions collected over different temporal intervals.

Unlike systems which collect garbage as a function of remaining space, JESQ must delete tokens on the basis of their semantics, since it is the presence/absence of tokens which trigger inferences in the system. Garbage collection is a general requirement for realtime expert systems that are intended to run indefinitely.

Accommodating Predetermined Inputs and Outputs

Consultative expert systems receive input from and submit output (advice) to a human being. With a cognitive component participating in the formulation of input and the interpretation of output, the choice of semantics and grain size for both inputs and outputs is a relatively flexible one. That is, the consultative framework provides the knowledge engineer with the flexibility to distribute the required domain reasoning between the user and the expert system as needed.

In contrast, JESQ is a performer, an active agent that receives input from environmental resources and directly manipulates them. This limits the semantics and grain size of expert system input and output to the those dictated by the resources in the environment. In general, inputs and outputs are more detailed and precise in JESQ's domain than those typical of consultative systems. They are low-level data (i.e., operating system responses and commands) rather than high-level opinions.

2.8 SUMMARY

JESQ and other YES/MVS domain expert systems represent a *technical* solution to a *management* problem. The hypothesis is that effectively managing a portion of the computer installation's activities

with expert systems provides the benefits of automation in spite of the lack of formalization, the complexity, and the dynamics that characterize the operations environment. This hypothesis may be extended to other process control domains.

However, the implementation of a continuous, realtime expert system that exerts active control over a real (as opposed to "toy") multi-agent environment is a relatively novel undertaking, introducing requirements not usually addressed in typical session-oriented consultative expert systems.

3. EXPERT SYSTEM ARCHITECTURE

This section describes the architecture of JESQ. Section 3.1 justifies the level of user modelling implemented, and outlines design objectives. Section 3.2 provides an overview of production systems, the skeletal framework in which JESQ is implemented, to allow the unfamiliar reader to understand the sections that follow. Section 3.3 describes the global organization of JESQ's knowledge base. A description of individual mechanisms in JESQ follows in section 3.4. Section 3.5 provides a detailed trace of JESQ's execution as it solves a JES queue space problem. Section 3.6 describes JESQ's position within the YES/MVS architecture and the operations environment.

3.1 LEVEL OF USER MODELLING

As discussed in section 2, the nature of the operations environment requires that an automated operator facility be easy to modify to reflect changing installation requirements and policies, and that the encoded policies be understandable to those involved in running the installation. These requirements have implications for the appropriate level of user modelling to implement in JESQ. This section examines the alternatives in the context of the operations management hierarchy, concluding that a surface level specification of the desired behavior best suits the requirements of the application.

A Framework for Analysis

The level of user modelling required can be analyzed in terms of the version of the installation management hierarchy depicted in figure 2. Modelling actions at the lowest level of the computer operator, the knowledge base encodes *surface behavior* as in the installation run book. Ideally, the computer operator follows procedures designed by the operations manager, responding to the conditions anticipated by performing pre-established sets of actions.

Modelling at the next level of the operations manager would involve an explicit representation of interactions in the operations environment, and representations for the semantics of "problems", "routine work", "the installation director's objectives", etc., culminating in the development of a knowledge base that encodes the principles by which managers *formulate* policy. The resulting code would output the surface behavior specification at the first level (the operator level). This program could be run in batch mode and the resulting policy description executed in realtime. Alternatively, these two levels could be combined in a system that monitors the target system, formulates policies, and executes them in realtime.

At the next level in the hierarchy, the operations director provides general guidelines for formulating operational policy. These guidelines include rules such as "Maximize throughput during prime shift" and "Give the Speech Group the best service where possible". Producing output at this level would involve representing higher level objectives such as "Allocate computing resources to groups involved in hot research areas" and "Take a utilitarian approach to resource allocation where other groups are concerned".

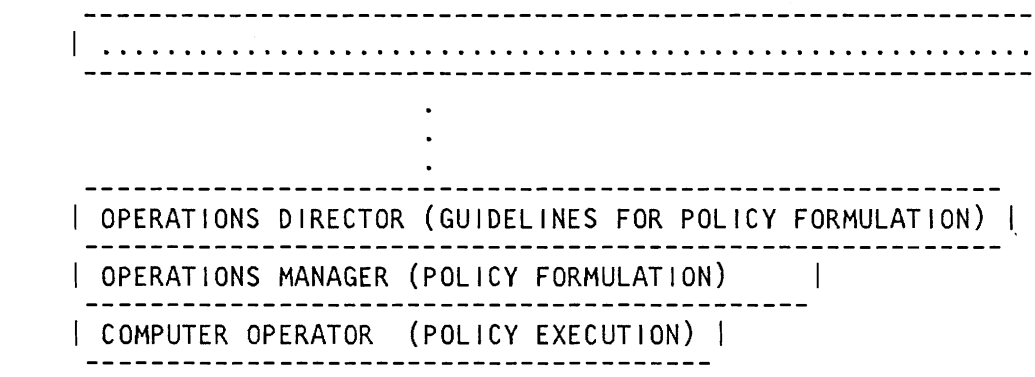


Figure 2: Operations Management Hierarchy

Of course, this conceptual framework can be extended indefinitely, although the rules become progressively more elusive and knowledge-intensive as we ascend the hierarchy. In some sense, the knowledge required at each level subsumes that of the levels subordinate to it. But these issues are beyond the scope of this thesis, and are presented here only to create a context for justifying the level of user modelling implemented in JESQ.

JESQ Models Surface Behavior

JESQ encodes knowledge of computer operations at the first level, that of the computer operator. Supposing that an industrial-strength facility that also encompasses knowledge at the second or higher levels could be developed at all, it can be assumed that the average installation could not effectively maintain such an abstract facility.

Combining levels is also discouraged in view of functional requirements and of the nature of the problem domain. As discussed in section 2, the operations environment is too complex to explicitly model, so encoding both policy formulation and execution knowledge does not represent a promising approach. Also, since the imposition of consistency is among the principle purposes of policy institution, it seems undesirable to formulate policies "on the fly" as a function of current values for state variables.

JESQ is therefore an automated operator, (rather than an automated operations manager or director) that directly executes operational policy. In effect, the goal of JESQ is to encode a portion of the run book and directly execute it. Within this context, the general goal of JESQ is to allow the installation to map policies into independent sets of rules wherever possible. This architecture is encouraged by the expert system shell upon which JESQ is built, and this tool is reviewed in the next section.

3.2 PRODUCTION SYSTEMS ARCHITECTURE

JESQ is implemented in a modified version of OPS5 [15], a general *production system* implemented in various dialects of LISP and in BLISS at Carnegie-Mellon University. OPS5 is intended as a tool for building applications in cognitive psychology, expert systems, and artificial intelligence [15]. Notable OPS-based expert systems include ACE [16], R1 [17], and XSEL [18]. While it was necessary to augment OPS5 with primitives for continuous realtime operation [19], its data-driven inference mechanism provided the basic framework required for the JESQ application and for others in

the YES/MVS domain. The purpose of this section is to provide a simplified overview of OPS5 that enables the reader to understand the implementation details described in the sections that follow.

There are essentially three components to the architecture of OPS5:

1. *Working Memory*: is a global database, typically containing an abstract representation of the state of the problem domain;
2. *Production Memory*: houses a set of IF ... THEN rules that encode problem domain expertise, expressed in terms of the domain representation in Working Memory;
3. *The Recognize/Act Cycle*: repeatedly applies the rules residing in Production Memory depending on the contents of Working Memory at any given moment.

The following sections describe these components in more detail.

Working Memory

Working Memory (WM) maintains the state of the problem domain and of the problem solving process. In JESQ, WM contains an abstract model of some of the components in the operations environment that impact JES queue space. JESQ's goals (e.g., reset the printer when queue space is restored) also reside in WM, along with housekeeping and control information.

WM is composed of *working memory elements* (wme). While OPS5 supports a few types of wmes, *attribute-value wmes* are the most prevalent in JESQ. An attribute-value wme is composed of an arbitrary *class name* and a set of *attribute-value pairs*. The attribute-value pairs express relations relevant to the wme class. For example, figure 3 depicts a wme from JESQ pertaining to the class *printer-status*. Attribute names are prefixed with the symbol ↑ and their values immediately follow, forming the attribute-value pairs. The attribute-value pairs in figure 3 indicate that printer PRT4 is currently AVailable, is loaded with CONFIDential forms, is currently printing job number 1234, and will not print any job that exceeds 30000 lines in length.

```
(printer-status
   ¬address      PRT4
   ¬status       AV
   ¬forms        CONFID
   ¬current-job  1234
   ¬line-limit   30000)
```

Figure 3: Sample JESQ Working Memory Element

An arbitrary number of printer-status wmes may concurrently reside in WM, each representing the status of a different printer (as identified by the ¬*address* attribute) in the environment.

In JESQ, such state variable wmes are updated periodically from responses to queries of the target system. These wmes appear in the antecedents of rules in Production Memory that make operational decisions based on the state of the target system. WM is highly dynamic, representing the state of the target system from moment to moment.

Production Memory

Production Memory (PM) is composed of IF ... THEN rules (or *productions*) that encode problem domain expertise. In JESQ, rules encode the problem-solving knowledge of operators, operations managers, systems programmers, and others involved in the management of JES queue space. PM is also called the *knowledge base* or *rule base*.

OPS5 productions consist of a *left hand side* (LHS) and a *right hand side* (RHS). The LHS is the antecedent of the rule and is composed of *condition elements*. The condition elements are essentially templates for matching against the contents of WM, and these can be coded at the desired level of abstraction. OPS5 provides constructs for specifying various match criteria (e.g., predicates, disjunctions, conjunctions) on wme values. Condition elements can also be negated. A LHS is considered to be satisfied when (1) there exist working memory elements that match all the non-negated condition elements, and (2) there exist no working memory elements that match the negated condition elements.

The RHS is the consequent of the rule and is composed of *actions* that are to be executed when the LHS is satisfied. Typical actions in JESQ include creating, modifying, and deleting wmes in WM, submitting queries to the target system, sending messages to the operator, and performing computations. It is also possible to call other programs from the RHS.

As an example, consider the simplified JESQ production in figure 4. The production is enclosed in parentheses. The symbol *p* identifies the construct as a production. The name of the production is *release-large-held-jobs*. The constructs that appear between the name and the symbol --> constitute the rule's LHS. It's condition elements are *processing-mode* (symbolically indicating a range of space left) and *wtr-queue-reply* (identifying the name of a job on the JES queue, the number of lines of output for that job, and whether or not it is being held). The match criteria specified include: (1) there is a queue space problem (poke, solve, or panic mode, discussed in section 3.3), and (2) there exists a held job that exceeds 30000 lines. The remaining constructs constitute the rule's RHS. The first action, *(remove 2)*, removes the wme that matches the second condition element in the LHS (i.e., the wtr-queue-reply) from WM. The second action transmits a command to the target system that releases the job in question (i.e., the job specified in the wme that matches the second condition element). Here, *<job>* is an unrestricted *variable* that will match any value associated with ¬*job-id* attribute.

Thus, this rule encodes the policy that all jobs with voluminous output that are being held on the queue should be released when a queue space problem exists. This will free the job for printing, DJ, or other fate that will, in turn, result in additional free queue space.

```
(p release-large-held-jobs
   (processing-mode
      ¬ mode << poke solve panic >>)
   (wtr-queue-reply
     ¬ print-lines > 30000
     ¬ job-id       <job>
     ¬ held?          Y)
-->
   (remove 2)
   (call remote-make release-held-job
      ¬ job-id <job>)
)
```

Figure 4: Sample JESQ Production (simplified)

Note that the rule is a *declarative* construct. In effect, the rule is a demon that will become active when its conditions are satisfied. Ideally, all program control resides in the interpreter's recognize-act cycle, although this is not always possible. (This is further discussed in section 5).

Recognize-Act Cycle

The recognize-act cycle controls the application of rules in PM, depending on the contents of WM. The cycle is transparent to the knowledge engineer. Ideally, the knowledge engineer need only encode knowledge of the problem domain in PM rules and implement an appropriate representation of the domain in WM. A simplified version of the OPS5 recognize-act cycle consists of three steps and a branch (see [15] for a complete discussion):

1. *Match*: Match the condition elements in the LHS of each rule against the current contents of WM. If no LHS are satisfied, stop. If one or more (the usual case) LHS are satisfied, continue.
2. *Conflict Resolution*: Given the satisfied rules and their associated wmes from step 1, select one to apply using general problem-solving heuristics. In particular, eliminate trivial infinite loops by prohibiting the repeated triggering of a rule by the same set of wmes, give preference to rules that match the most recently created wmes, and to those rules that specify the most LHS tests.
3. *Act*: Execute the actions of the rule selected in step 2. The actions typically change WM (adding, deleting, or modifying wmes), so that on the next iteration of the cycle, a different set of LHS is satisfied.
4. Go to step 1.

JESQ and other YES/MVS domain expert systems employ a modified version of this cycle, including a step that reduces the input to conflict resolution according to prespecified rule groupings (see section 3), and a step that picks up messages from the target system. The resulting cycle is as follows:

1. *Match*
2. *Conflict Set Reduction* by priority (rule group)

3. (OPS5) *Conflict Resolution*

4. *Act*

5. *Pickup* target system messages

6. Go to step 1.

This too is an oversimplified description. Several other modifications have also been implemented, including a function that causes the system to idle (rather than stop) when no LHS are satisfied. See [19] for implementation details.

With the necessary background provided, the next section describes the JESQ knowledge base.

3.3 JESQ KNOWLEDGE BASE ORGANIZATION

The organization of JESQ's knowledge base reflects an objective to achieve a simple representation that provides for mapping the elements of queue space management policy into generic classes of rules. To accomplish this, rules in JESQ are grouped along two orthogonal dimensions: by function (e.g., query submission, information collection) and by problem severity (as a function of space left on the queue). That is, each rule is labelled as belonging to functional group (by its *task* condition element) and as being applicable in some range of space left (by its *processing-mode* condition element). This architecture is intended to standardize the method by which the knowledge base is maintained. That is, the process of adding a new policy to the knowledge base can usually be accomplished by adding the appropriate rules to each of the functional rule groups and specifying the level of problem severity under which the policy may be applied. In short, the process of augmenting JESQ with a new policy is intended to be algorithmic. This architecture is further described in the following sections.

Rule Grouping by Function

JESQ rules are grouped into functional classes. The computational consequences of this grouping scheme have to do with the relative priorities of individual rules in the context of the conflict resolution step of the recognize-act cycle. Each functional rule group has been assigned a priority, and that priority is used to determine which rule will be invoked when rules from more than one group are concurrently satisfied in a given iteration of the cycle. In effect, resolving rule application conflicts with this mechanism serves to augment the existing OPS5 conflict resolution algorithm. The set of satisfied rules is first reduced to contain only those of the same functional group, and the resulting set is resolved (by OPS5 conflict resolution) on the basis of recency of information and specificity of LHS conditions.

JESQ includes the following functional rule groups. All rule groups work cooperatively to support the Knowledge-Based Action group. This last group encodes operational policy at its highest level.

- *System Initialization and Control*: This group contains rules that create the abstract internal model of the target system environment. These rules are general rather than configuration-specific. The model building rules are triggered by wmes derived from the dataset that describes the installation's configuration to the target MVS operating system, making JESQ more transportable to other installations. Rules in this group also implement meta-level control of rule groups as a function of the severity of the queue space problem at hand, as described in the next section. This rule group also suppresses certain actions when specified by the operator.

- *Wait*: This group contains one rule which causes JESQ to wait until an interrupt is generated by either the receipt of a target system message or by an internally scheduled event.

- *Periodic Query Submission and Timeout Handling*: This group controls the periodic querying of target system resource states. Query intervals are based on estimates of the reliability over time of the information being captured. These intervals are adjusted dynamically by the expert system as it executes, based on the severity of the problem at hand and on the expert system's know-

ledge of its own actions. For example, JES queue space is monitored every five minutes in the normal case and every 50 seconds when a problem is detected, since JESQ's actions will usually free space quickly in a problem situation. Rules are also included to resubmit queries that have been lost in transmission (i.e., timed out).

- *Information Collection and Data Reduction/Expansion*: This group includes rules that collect target system messages and update JESQ's internal model accordingly. Portions of this abstract model appear in the LHS of the Knowledge-Based Action rules that make deductions and take action. Some rules in this group map a single response wme into a single internal model wme. (Target system responses are translated to wme format by MCCF, a supporting facility described in section 3.6). Other rules perform data reduction, manipulating multiple response wmes to produce a single summary wme that is referenced by the Knowledge-Based Action rules. Still other rules perform data expansion, supplying attributes with values that are only implied by target system responses.

- *Miscellaneous Cleanup and Response Collision Collection*: This rule group deletes target system responses and expert system-generated goals from WM. Rules in this group also delete asynchronously arriving responses to duplicate queries that have been delayed by failing or sluggish target system resources.

- *Knowledge-Based Action*: All rules in the above described groups exist to support the Knowledge-Based Action rules which encode queue space management policy. The policies encoded mirror those described in the Domain Overview. In particular, rules are included to protect the remaining queue space, to set up for future actions, to reset target system parameters when space returns to acceptable levels, to free queue space when a problem exists, and to alert the operator to potential problems that cannot be further diagnosed without additional information. These rules are further decomposed into three subgroups of varying priority: *low-, medium-,* and *high-priority-knowledge-based-actions.*

Priorities have been assigned to functional rule groups as follows. In general, information collection rules dominate all others, guaranteeing that expert system actions reflect the most recent target system information available. System control rules have the next highest priority, since these partition knowledge-based actions and their supporting processes. The three groups of knowledge-based actions have the next highest set of priorities. Cleanup rules dominate in lieu of the applicability of the rule groups described thus far, allowing the query submission rules of next lower priority to fire upon the removal of old responses. Naturally, the idle action is of lowest priority. Some examples of rules in these groups appear in section 3.4.

Rule Grouping by Problem Severity

Groups of rules are enabled/disabled dynamically during expert system execution according to the severity of the queue space problem at hand. For example, a drastic action such as varying the main processor offline is appropriate when only 3% of the queue space remains, but not when 10% remains. To implement this knowledge, thresholds of space left are mapped to five symbolic *processing modes* (NORMAL, WATCH, POKE, SOLVE, and PANIC), each associated with range of space left on the JES queue. Some actions are limited to a single processing mode (e.g., varying the main processor offline). Other actions span multiple processing modes (e.g., raising the line limit on a printer).

The processing mode scheme implements meta-level control, suppressing sets of actions as a function of problem severity (i.e., space left on the queue). (In reality, transitions from mode to mode are more complex than this description might imply. A discussion of this mechanism is included in the following section).

3.4 JESQ MECHANISMS AND DETAILED DESIGN ISSUES

This section illuminates some of the mechanisms that implement the functions described in the last section.

Initialization

JESQ is initiated by the appearance of wmes in WM with class name *task*. A task wme exists for each functional rule group, identifying that group to the underlying priority mechanism (see [19] for details). The rule shown in figure 5 (condensed for brevity) creates a base set of wmes, triggering the execution of rules which submit initial queries, create the internal model of the target system, and perform other initialization tasks. Thus, this rule is the first in a series of boot-strapping rules.

Referring to figure 5, wmes such as *jes-q-space*, *sys1* (the main processor), and *dj-job* (the job that performs the transfer of jobs from the JES queue to tape) are portions of the abstract model of the target system. Values for the attributes of these wmes are later derived from responses to queries of the target system. The *periodic-query* wme is essentially a table entry that is referenced by the query submission rules, providing the interval between queries of the indicated type. This interval is dynamically adjusted by JESQ at execution time, depending on the state of the problem-solving process.

```
(p initialization
   (task
     ¬task-id  system-control)
-->
   (make processing-mode
     ¬mode normal)
   (make jes-q-space)
   (make sys1)
   (make dj-job)
   (make periodic-query
     ¬type  jes-q-space
     ¬interval  5
     ¬unit  min)

    ---etc.----

)
```

Figure 5: Initialization Rule

Since JESQ may be initiated at any time, nothing is assumed about the state of the target system at initialization time. Initialization queries are submitted to request values for system parameters that vary by shift, by problem state, and other conditions.

Table-Driven Target System Model Creation

Associated with each relevant component in the installation hardware configuration are a set of wmes that maintain the pertinent characteristics of that component in JESQ's internal model of the target system. For example, for each printer in the environment (as indicated by wmes derived from the file that describes the configuration to the operating system) JESQ maintains a *printer-status* wme (for recording the status of the printer from moment to moment), a *printer-queue* wme (for recording line totals of the jobs queued for the printer), and others. Figure 6 depicts an abbreviated version of the rule that creates these entities. Upon detection of a *printer* wme created by the configuration description program, this rule *make*s the needed wmes in WM. Similar rules exist to create wmes associated with other devices in the environment.

```
(p make-printer-entities
   (task
     ¬task-id system-control)
   (printer
     ¬ address <printer>)
-->
   (make printer-status
     ¬ reliable? no
     ¬ address <printer>
     ¬ status none
     ¬ forms none
     ¬ current-job none
     ¬ line-limit none)
   (make printer-queue
     ¬ reliable? no
     ¬ address <printer>
     ¬ confid-lines none
     ¬ ald-lines none
     ¬ onepart-lines none
     ¬ short-lines none
     ¬ confids-lines none
     ¬ total-lines none
     ¬ small-job-lines none
     ¬ medium-job none
     ¬ big-job none)

    ---etc.----

)
```

Figure 6: Table-Driven Printer Model Creation

Note that these rules are *table-driven* [20], executing for each conceptual table entry created by the configuration description program. For example, the rule shown in figure 6 will fire for each occurrence of the *printer* wme in WM, so that the rule base need not be changed with the introduction of new printers in the environment (although a configuration change may result in modification of the associated operational policy). Rather, this change is isolated to the configuration description program.

Rule Group Control

As discussed in section 3.3, the *processing-mode* wme implements meta-level control of rule groups, enabling/disabling sets of rules as a function of space left on the JES queue. A condition element that

matches the processing-mode wme appears in the LHS of all rules in JESQ that are problem state dependent.

In general, processing mode transitions are a function of static, mutually-exclusive range specifications. However, consider the case where the queue space remaining fluctuates within a few percentage points on the boundary of the definition of two processing modes. (Fluctuation within a few percent is common; queue space jumps by leaps and bounds only when significant actions are taken by some environmental agent). Since an action initiated in one mode may be terminated in its boundary mode, it is unacceptable to perform such actions when queue space is fluctuating on the boundary of their definition. To illustrate, consider the case where the remaining queue space fluctuates between 18% and 20%. In terms of the absolute ranges by which processing mode is defined, JESQ would be fluctuating between *poke* (15%-19%) and *watch* (20%-24%) modes. In poke mode, JESQ raises the limit on the 3211 printer (conditions permitting), allowing larger jobs to print. In watch mode, this line limit is reset to its original value, since queue space is at a more comfortable level. Were a mechanism not included to prevent it, JESQ would set and reset the line limit repeatedly as space left fluctuated between watch and poke modes. Clearly, this could not be called "intelligent" behavior.

To avoid this potential problem, JESQ implements a sort of histirisis curve to prevent frequent processing mode changes on the boundary of their definition. Downward transitions are performed as a function of absolute ranges. Transitions from modes to non-upward-adjacent modes are similarly performed. However, the transition from a mode to its upward-adjacent mode requires the achievement of a percentage of space left that *exceeds its defined lower bound.* This is illustrated in figures 7 and 8.

The rule in figure 7 effects a transition from any mode other than poke (i.e., normal, solve, or panic) to watch mode (the <<double angle brackets>> implement a disjunction). The range used here is 20%-24%. As described, the rule in figure 8 effects a transition from poke mode to watch mode, requiring the achievement of at least 22% (not 20%) space left. This implementation would prohibit

the fluctuation between modes that would cause the unintelligent manipulation of the model 3211 printer in the example situation described above.

```
(p set-to-watch-mode-from-non-upward-adjacent-modes
   (task
     ¬task-id  system-control)
   (processing-mode
     ¬mode  << normal solve panic >>)
   (jes-q-space
     ¬percent-left  { >= 20  <= 24 })
-->
   (modify 2
     ¬mode watch)
)
```

Figure 7: Processing Mode Transition From a Non-upward-adjacent Mode

```
(p set-to-watch-mode-from-poke-mode
   (task
     ¬task-id  system-control)
   (processing-mode
     ¬mode  poke)
   (jes-q-space
     ¬percent-left  { >= 22  <= 24 })
-->
   (modify 2
     ¬mode watch)
)
```

Figure 8: Processing Mode Transition From an Upward-adjacent Mode

Of course, it is theoretically possible to produce such "unintelligent" behavior despite this mechanism. For instance, queue space might have fluctuated between 18% and *22%* in the example situation, allowing the behavior that this mechanism is intended to prevent. However, the ranges coded are based on operators' empirical observations, and failure of the mechanism should rarely occur. Again, this is because fluctuation in space left is characterized by a certain "locality".

Periodic Action Control

Periodic actions are common in the operations environment. An obvious example, JESQ must periodically query the status of JES queue space. The rule that accomplishes this is shown in figure 9. The *periodic-query* wme gives the interval of query submission (e.g., 5 minutes), and is always present in WM. The *query-request* wme is the token that actually triggers the rule, provided that the response (*jes-q-reply*) to the last query is not still in WM (note the application of negated condition elements)

waiting to be processed (by an information collection rule). When the rule is invoked, the call to *remote-make* sends the queue space query to the target system, and the call to *timed-make* specifies the reappearance of the *query-request* token in the interval specified by the *periodic-query* wme. Finally, the *(remove 3)* action deletes the *query-request* wme that triggered the rule this time.

```
(p send-jes-space-query
   (task
     ¬task-id  query-submission)
   (periodic-query
     ¬type  jes-q-space
     ¬interval  <iv>
     ¬unit  <un>)
   (query-request ¬type jes-q-space)
  -(jes-q-reply)
-->
   (call remote-make jes-queue-space-query)
   (call timed-make query-request
         ¬type jes-q-space  (in <iv> <un>))
   (remove 3)
)
```

Figure 9: Periodic Action Control

Thus, this rule is periodically invoked without end, once each time the *query-request* token appears. In general, each periodic action in JESQ is implemented in an isolated rule that employs this mechanism.

Response Timeout Handling

Responses to queries of the target system can be lost enroute to JESQ. In such cases, the facility that passes responses to JESQ (MCCF) sends instead the time-out message (wme) of the programmer's choice. The choice in this case is a token that triggers the resubmission of the lost query. While the *query-request* wme serves this purpose, the rule it triggers (figure 9) provides for the re-creation (via timed-make) of the same token at a later time, causing additional queries to be generated. Since this is unacceptable, an additional rule is coded (figure 10) that merely resubmits the query, without specifying the future generation of additional tokens.

```
(p send-isolated-jes-space-query
   (task
     ¬task-id  query-submission)
   (isolated-query-request ¬type jes-q-space)
-->
   (call remote-make jes-queue-space-query)
   (remove 2)
)
```

Figure 10: Response Timeout Handling

The time-out token *isolated-query-request* triggers this rule once and is removed from WM.

Abstract Model Update

Responses to queries received from the target system are reflected in the appropriate portions of JESQ's internal model of the environment and deleted. Figure 11 depicts the rule that updates the *jes-q-space* wme to reflect the percentage of space left specified by the *jes-q-reply* response to the last JES queue space query.

```
(p jes-q-space-update
   (task
     ¬task-id information-collection)
   (jes-q-reply  ¬percent-left  <pl>)
   (jes-q-space)
-->
   (remove 2)
   (modify 3
      ¬percent-left  <pl>)
)
```

Figure 11: Abstract Model Update

Similar rules exist to maintain other portions of JESQ's internal model.

Accounting for Manual Operator Action

JESQ generates goals for itself to reset environmental parameters that it has manipulated to free queue space over the course of solving a problem. It is necessary to generate explicit goals in order to distinguish between JESQ's having altered the parameter and the operator's having done so. Should the operator have altered the parameter manually, it is inappropriate for JESQ to restore it to

its usual value, since the reasons for the operator's action are beyond JESQ's knowledge sources (otherwise, JESQ itself would have altered the parameter). For example, JESQ disables the main processor in panic mode to prevent the initiation of additional jobs that might generate output on the JES queue. In this case, JESQ generates *(goal ¬type vary-sys1-back-online)* to trigger the inverse action once space has improved (i.e., normal, watch, poke, or solve mode has been achieved). However, were the human operator to disable the main processor under conditions of normal queue space, say, for hardware maintenance, it would obviously be inappropriate for JESQ to automatically enable it. Thus, JESQ records its own actions in cases where reset actions are appropriate once queue space has improved.

Now consider the case where JESQ has disabled the main processor in a problem state, the problem has been resolved, but the operator has enabled the main processor himself. In this case, JESQ must remove its goal to avoid subsequent confusion and to free space in WM. The rule that accomplishes this appears in figure 12.

```
(p account-for-operator:already-varied-sys1-online
   (task
      ¬ task-id cleanup)
   (processing-mode
      ¬ mode << normal watch poke solve >>)
   (sys1
      ¬ status ONLINE)
   (goal
      ¬type vary-sys1-back-online)
-->
   (remove 4)
)
```

Figure 12: Accounting for Manual Operator Action

This rule removes the goal to re-enable the main processor, provided that: (1) the JES queue is not in a panic state (processing-mode), (2) the main processor has already been enabled (is ONLINE), and (3) the goal to do so still exists in WM.

Idle Control

JESQ idles when no "substantive rules" in its knowledge base are applicable. The idle state may be interrupted by the receipt of a target system message or by an internally scheduled event. This is implemented as shown in figure 13.

```
(p wait
   (task
     ¬task-id wait)
-->
   (modify 1)
   (call ops-wait)
)
```

Figure 13: Idle Control

The *wait* task has the lowest priority in the knowledge base. When no other rules are satisfied, the idle function, *ops-wait*, is invoked. Recalling from section 3.2 that the OPS5 interpreter will only once execute a given rule in conjunction with a given set of wmes, the *(modify 1)* action is needed to renew the wait *task* wme so that the system can again idle when no other rules are satisfied. Omission of the *modify* action would allow JESQ to idle only once over the course of its execution.

Having described the architecture of JESQ and the OPS5 language, the next section depicts an OPS5 trace of JESQ as it solves a JES queue space problem.

3.5 JESQ DEMONSTRATION TRACE

This section provides a demonstration of JESQ solving a queue space problem.

Instructions for Reading the Trace

The trace shown here is output by the modified OPS5 interpreter, and is never seen by the computer operator. Wmes of interest have been displayed between rule firings in the following demonstration, and these are depicted as in previous sections, along with their *time tags* which are assigned by the interpreter for purposes of identification. The firing of any single rule is indicated by a line of interpreter output of the form:

```
a. b                    c
```

```
where a = the number of rules that have fired thus far
      b = the name of the rule fired
      c = the time tags of the wmes bound to the rule's LHS
```

The operator sees only the screen images labelled as such in the demonstration that follows. A hierarchical collection of screens is provided to the operator. The TOP LEVEL screen contains one descriptive line of text per JESQ action or suggestion. For most actions, a detailed screen is provided which may be accessed by positioning the cursor at the message of interest on the TOP LEVEL screen and pressing a key on the terminal. The detailed screen provides a description of the current situation, a corrective operating system command or manual action to be executed, and an explanation regarding its appropriateness. For brevity, only one such detailed screen is shown in the following demonstration.

Demonstration

Upon initiating JESQ, initialization rules are fired which create portions of the internal model of the environment:

```
1. JM:INITIALIZATION 7
2. JM:MAKE-PRINTER-ENTITIES 7 19
3. JM:MAKE-PRINTER-ENTITIES 7 18
4. JM:MAKE-TAPE-DRIVE-ENTITIES 7 17
5. JM:MAKE-TAPE-DRIVE-ENTITIES 7 16
6. JM:MAKE-TAPE-DRIVE-ENTITIES 7 15
```

For example, for each tape drive (as indicated by the *tape-drive* wme) in the environment, a *tape-drive-status* wme is created with null values for attributes:

```
17:  (TAPE-DRIVE     ¬ADDRESS 282)
16:  (TAPE-DRIVE     ¬ADDRESS 281)
15:  (TAPE-DRIVE     ¬ADDRESS 280)

44:  (TAPE-DRIVE-STATUS     ¬RELIABLE? NO     ¬ADDRESS 280     ¬STATUS NONE
)
42:  (TAPE-DRIVE-STATUS     ¬RELIABLE? NO     ¬ADDRESS 281     ¬STATUS NONE
)
40:  (TAPE-DRIVE-STATUS     ¬RELIABLE? NO     ¬ADDRESS 282     ¬STATUS NONE
)
```

These wmes then trigger the submission of initial queries regarding relevant entities in the environment:

```
7. JM:SEND-PRINTER-STATUS-QUERY 2 22 35 10
PRINTER-STATUS-QUERY PRT1
8. JM:SEND-PRINTER-STATUS-QUERY 2 22 30 10
PRINTER-STATUS-QUERY PRT4
9. JM:SEND-DJ-STATUS-QUERY 2 10 28
DJ-STATUS-QUERY
10. JM:SEND-SYS1-STATUS-QUERY 2 10 27
SYS1-STATUS-QUERY
11. JM:SEND-JES-SPACE-QUERY 2 20 25 10
JES-QUEUE-SPACE-QUERY
```

Responses are translated to wme format by MCCF (see section 3.6) and placed in WM:

```
51:  (PRINTER-STATUS-REPLY     ¬MESSAGE-ID IAT8562     ¬ADDRESS PRT4     ¬S
TATUS AV     ¬FORMS 1PART     ¬LINE-LIMIT NONE     ¬CURRENT-JOB NONE)
50:  (PRINTER-STATUS-REPLY     ¬MESSAGE-ID IAT8562     ¬ADDRESS PRT1     ¬S
TATUS OFF     ¬FORMS 1PART     ¬LINE-LIMIT 30000     ¬CURRENT-JOB NONE)
```

These responses typically supply values for the attributes of internal model wmes:

```
34:  (PRINTER-STATUS     ¬RELIABLE? NO     ¬ADDRESS PRT1     ¬STATUS NONE
  ¬FORMS NONE     ¬LINE-LIMIT NONE     ¬CURRENT-JOB NONE)
29:  (PRINTER-STATUS     ¬RELIABLE? NO     ¬ADDRESS PRT4     ¬STATUS NONE
  ¬FORMS NONE     ¬LINE-LIMIT NONE     ¬CURRENT-JOB NONE)
```

The receipt of such responses trigger information collection rules that update the corresponding portions of the internal model. Below, the receipt of a PRINTER-STATUS-REPLY wme for each printer triggers the execution of rules which update the corresponding PRINTER-STATUS wmes:

```
12. JM:PRINTER-STATUS-UPDATE 8 51 29
13. JM:PRINTER-STATUS-UPDATE 8 50 34

57:  (PRINTER-STATUS    ¬RELIABLE? YES    ¬ADDRESS PRT1    ¬STATUS OFF
  ¬FORMS 1PART    ¬LINE-LIMIT 30000    ¬CURRENT-JOB NONE)
54:  (PRINTER-STATUS    ¬RELIABLE? YES    ¬ADDRESS PRT4    ¬STATUS AV
 ¬FORMS 1PART    ¬LINE-LIMIT NONE    ¬CURRENT-JOB NONE)
```

Similarly, the response to the query regarding the DJ job (which allows the copying of jobs from the queue to tape) arrives, triggering the corresponding information collection rule, and the DJ-JOB wme is updated to reflect that the DJ job is currently inactive:

```
14. JM:DJ-STATUS-UPDATE:FROM-QUERY:DJ-NOT-FOUND 8 14 58

60:  (DJ-JOB    ¬STATUS INACTIVE)
```

Next, the response to the query regarding the status of the main processor (SYS1) arrives, and the SYS1 wme is updated to reflect that the main processor is currently online:

```
15. JM:SYS1-STATUS-UPDATE:FROM-QUERY 8 13 62

64:  (SYS1    ¬STATUS ONLINE)
```

Finally, the response to the query regarding the status of JES queue space arrives, and the JES-Q-SPACE wme is updated to reflect that 72% of total space remains free:

```
16. JM:JES-Q-SPACE-UPDATE 8 66 12

69:  (JES-Q-SPACE    ¬PERCENT-LEFT 72)
```

This is the normal situation. JESQ will now wait until it is time to submit another set of queries:

```
17. JM:WAIT 1
```

That time arrives in the trace below, and another set of routine queries is generated:

```
18. JM:SEND-PRINTER-STATUS-QUERY 2 22 73 10
PRINTER-STATUS-QUERY PRT4
19. JM:SEND-JES-SPACE-QUERY 2 20 75 10
JES-QUEUE-SPACE-QUERY
20. JM:SEND-PRINTER-STATUS-QUERY 2 22 72 10
PRINTER-STATUS-QUERY PRT1
```

The JES queue space query response arrives, and this triggers two rules: JESQ's internal model of the environment is updated to reflect that now only 3% space left remains on the queue, and JESQ sets itself to panic mode:

```
21. JM:JES-Q-SPACE-UPDATE 8 78 69
22. JM:SET-TO-PANIC-MODE-FROM-NORMAL 7 11 81

81:  (JES-Q-SPACE    ¬PERCENT-LEFT 3)
```

This triggers JESQ's first corrective action. JESQ varies the main processor offline to block the initiation of additional jobs that may create output on the JES queue:

```
23. JM:VARY-SYS1-OFFLINE 6 83 64 85 10
```

The operator receives his first message on the TOP LEVEL screen:

```
      YES/MVS TOP LEVEL                                  Pending:   0
===>


        VARY SYS1 OFFLINE









PF01 PF02 PF03 PF04  PF05  PF06 PF07 PF08 PF09 PF10  PF11  PF12
     U.I. EXIT WRKNG SELCT DONE BACK FWD       RFRSH ERRST HOME
```

The operator can display the associated detailed screen as previously described:

```
    YES/MVS AUTHORIZE                                   Pending:   0
===>
TOP-LEVEL MSG: VARY SYS1 OFFLINE

SITUATION:   JES queue space is at the panic level. SYS1
             is online, allowing additional jobs to be
             initiated and further deplete the remaining
             space. The system will crash if more output
             is generated.

EXPLANATION: SYS1 must be varied offline in order to
             protect the little space that remains.
             Press U-DO, I-DID, or NO-DO.


MVS COMMAND: 8f vary sys1,offline



PF01 PF02 PF03 PF04  PF05  PF06 PF07 PF08 PF09 PF10  PF11  PF12
          EXIT       U-DO               I-DID NO-DO ERRST HOME
```

There are two modes of operation. In "advisory mode", the operator authorizes expert system actions on a command by command basis via terminal keys (U-DO, I-DID, NO-DO) on the detailed screen shown above. In "authorized mode", actions not requiring manual intervention are taken automatically, so the screens shown represent recent expert system actions. Of course, actions requiring manual intervention (e.g., mounting a tape) are always advised regardless of the mode of execution.

Upon varying the main processor offline, JESQ receives a response from the operating system which triggers a rule that updates the SYS1 wme. JESQ also generates a GOAL wme for itself to vary the main processor back online once the state of the JES queue has improved:

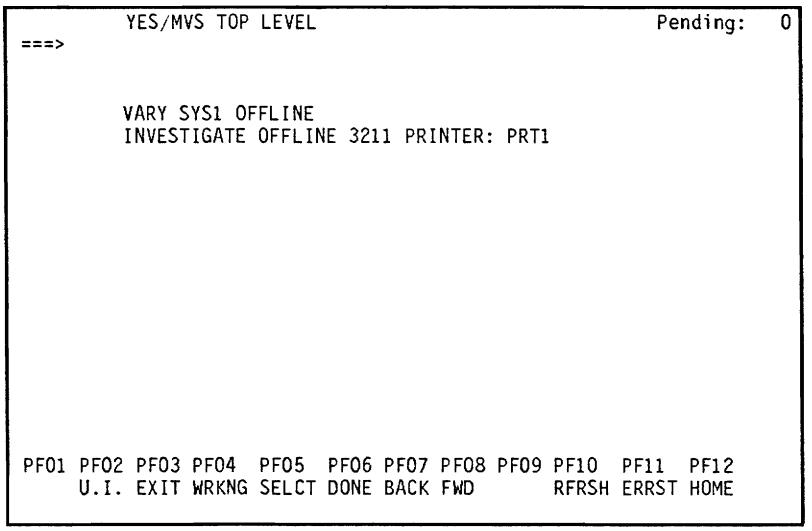
```
24. JM:SYS1-STATUS-UPDATE:FROM-VARY 8 64 88

90:  (SYS1    ¬STATUS OFFLINE)

87:  (GOAL    ¬TYPE VARY-SYS1-BACK-ONLINE)
```

Next, JESQ adjusts the JES queue space query interval (causing queries to be submitted more often), since its actions should cause rapid changes in queue space status:

```
25. JM:SET-FAST-JES-QUERY-INTERVAL 5 84 20
```

JESQ then notices that a printer is offline in the midst of a queue space problem, prohibiting jobs from printing and being deleted from the queue. Of course, JESQ could vary the printer online automatically, but an operator could be changing paper or performing some other task that renders it dangerous to do so. Instead, JESQ sends a message to the operator's console, alerting him to the offline printer condition:

```
26. JM:OFFLINE-PRINTER-CHECK 4 83 18 57 10
```



Next, JESQ submits a batch of queries that support other problem-state knowledge-based actions. In particular, JESQ searches for a tape drive on which to start DJ, requests a list of the jobs queued for printers in the environment, and a list of the jobs that are being held on the queue by online terminal users:

```
27. JM:SEND-TAPE-DRIVE-STATUS-QUERY 2 83 21 43 60 10
TAPE-DRIVE-STATUS-QUERY 280
28. JM:SEND-TAPE-DRIVE-STATUS-QUERY 2 83 21 41 60 10
TAPE-DRIVE-STATUS-QUERY 281
29. JM:SEND-TAPE-DRIVE-STATUS-QUERY 2 83 21 39 60 10
TAPE-DRIVE-STATUS-QUERY 282
30. JM:SEND-PRINTER-QUEUE-QUERY 2 83 23 37 10
PRINTER-QUEUE-QUERY PRT1
31. JM:SEND-PRINTER-QUEUE-QUERY 2 83 23 32 10
PRINTER-QUEUE-QUERY PRT4
32. JM:SEND-HOLD-QUEUE-STATUS-QUERY 2 83 24 26 10
HOLD-QUEUE-STATUS-QUERY
```

The list of jobs (set of wmes) queued for the 3211 printer (PRT1) happen to arrive first, one wme for each job:

```
103:  (WTR-QUEUE-REPLY    ¬MESSAGE-ID IAT8131    ¬DEVICE PRT1    ¬JOB-ID
 3000    ¬FORMS 1PART    ¬DEVICE-TYPE PRT    ¬HELD? N    ¬PRINT-LINES 40
000)
102:  (WTR-QUEUE-REPLY    ¬MESSAGE-ID IAT8131    ¬DEVICE PRT1    ¬JOB-ID
 2000    ¬FORMS 1PART    ¬DEVICE-TYPE PRT    ¬HELD? N    ¬PRINT-LINES 10
0)
101:  (WTR-QUEUE-REPLY    ¬MESSAGE-ID IAT8131    ¬DEVICE PRT1    ¬JOB-ID
 1000    ¬FORMS 1PART    ¬DEVICE-TYPE PRT    ¬HELD? N    ¬PRINT-LINES 10
0)
```

These responses are then analyzed by a group of rules that compute line totals by printed-form, classify the jobs by size, and produce a summary wme for the queue. Meanwhile, the operator has

varied the 3211 printer online, and the response is captured by JESQ, triggering the update of the PRINTER-STATUS wme as well:

```
33. JM:RECORD-MEDIUM-JOBS 9 103 38
34. JM:COUNT-ONEPART-LINES 9 106 108
35. JM:COUNT-TOTAL-LINES 9 110 112
36. JM:COUNT-ONEPART-LINES 9 102 116
37. JM:COUNT-SMALL-JOB-LINES 9 118 120
38. JM:COUNT-TOTAL-LINES 9 122 124
39. JM:COUNT-ONEPART-LINES 9 101 128
40. JM:COUNT-SMALL-JOB-LINES 9 130 132
41. JM:COUNT-TOTAL-LINES 9 134 136
42. JM:CLEANUP-WTR-QUEUE-REPLY:COUNTED-JOBS 9 138
43. JM:CLEANUP-WTR-QUEUE-REPLY:COUNTED-JOBS 9 126
44. JM:CLEANUP-WTR-QUEUE-REPLY:COUNTED-JOBS 9 114
45. JM:PRINTER-QUEUE-UPDATE 9 140 36
46. JM:PRINTER-STATUS-UPDATE 8 104 57
```

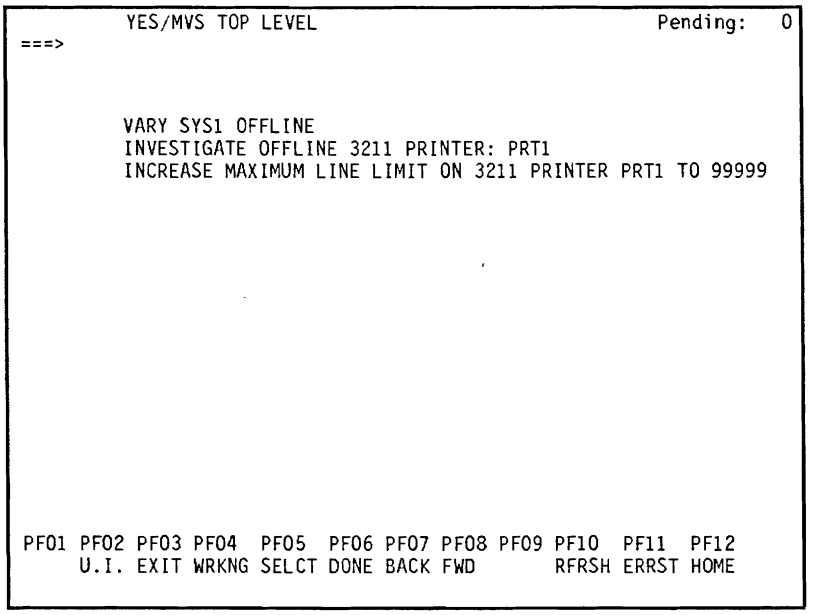
Referring to the internal model wmes shown below, the situation is as follows: JES queue space is low, the 3211 printer is set to print only jobs not exceeding 30000 lines (¬LINE-LIMIT 30000), the printer is about to idle with this setting (¬SMALL-JOB-LINES 200), and there is at least one job which exceeds this limit (¬MEDIUM-JOB 3000) waiting to print:

```
150:  (PRINTER-STATUS    ¬RELIABLE? YES    ¬ADDRESS PRT1    ¬STATUS AC
  ¬FORMS 1PART    ¬LINE-LIMIT 30000    ¬CURRENT-JOB 1234)

147:  (PRINTER-QUEUE    ¬RELIABLE? YES    ¬ADDRESS PRT1    ¬CONFID-LINES
 0    ¬ALD-LINES 0    ¬ONEPART-LINES 40200    ¬SHORT-LINES 0    ¬CONFIDS
-LINES 0    ¬MEDIUM-JOB 3000    ¬BIG-JOB NONE    ¬SMALL-JOB-LINES 200
 ¬TOTAL-LINES 40200)
```

Upon recognition of this situation, JESQ increases the line limit to allow job 3000 (and possibly others) to print, freeing queue space:

```
47. JM:INCREASE-3211-MAXLINES 6 83 18 150 147 10
```

And the operator receives a high level message on the expert system console:



As in the case of the main processor, JESQ also generates a goal to reset this line limit once queue space improves:

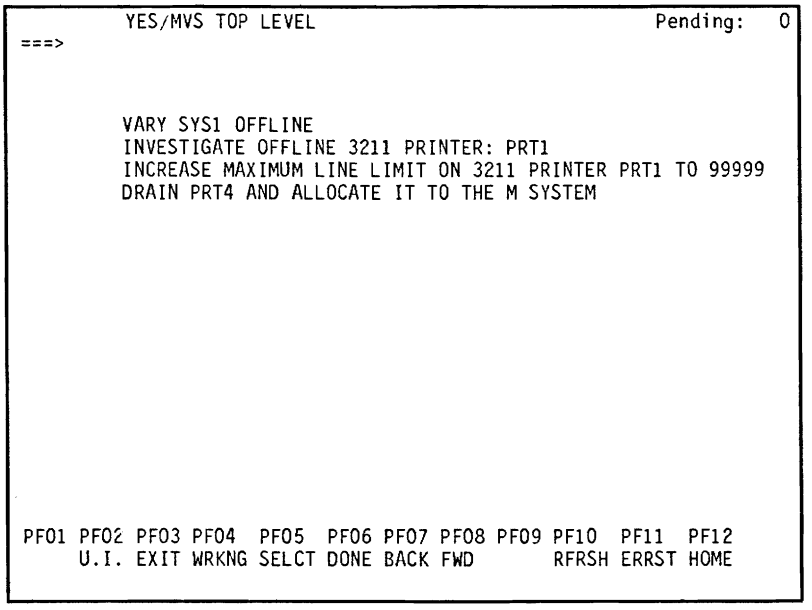
```
155:  (GOAL    ¬TYPE RESET-3211-MAXLINES    ¬ADDRESS PRT1)
87:  (GOAL    ¬TYPE VARY-SYS1-BACK-ONLINE)
```

Of course, raising the line limit will not *solve* the queue space problem (a 40000 line job did not bring queue space down to 3%). Rather, this action will merely buy time while JESQ continues to search for the real culprit.

Next, the response to the query regarding the status of the model 3800 printer arrives. This printer is shared between the target system and other systems in the environment. Upon recognition that the printer is allocated to another system, JESQ snatches the printer in case a use for it is found during the remainder of the problem solving process:

```
48. JM:PRINTER-STATUS-UPDATE 8 156 54
49. JM:RESERVE-THE-3800-FOR-LATER-ACTION 5 83 19 159 10
```

As usual, the operator receives a message on the expert system console:



Next, responses to tape drive queries begin to arrive, triggering rules that update the corresponding portion of JESQ's internal model of the environment:

```
50. JM:TAPE-DRIVE-STATUS-UPDATE 8 164 40
51. JM:TAPE-DRIVE-STATUS-UPDATE 8 163 42
52. JM:TAPE-DRIVE-STATUS-UPDATE 8 162 44
```

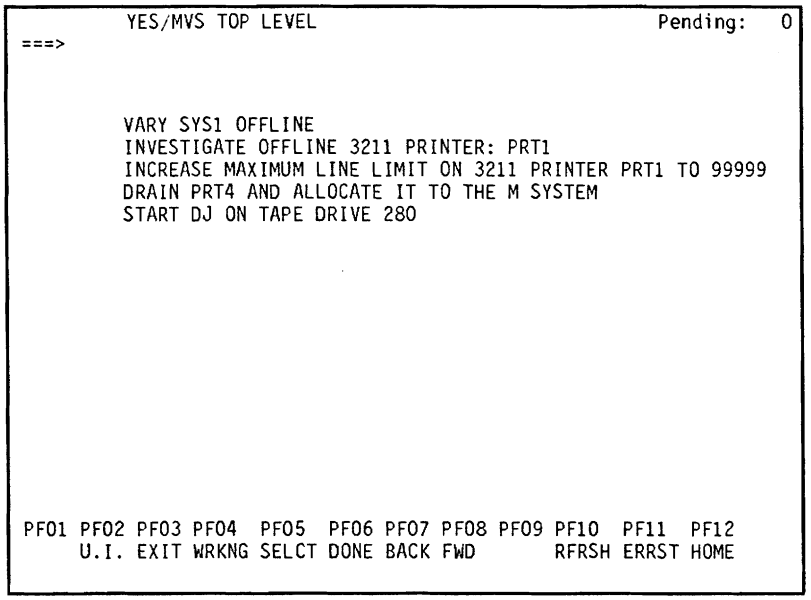
Tape drives 281 and 282 are unavailable (¬STATUS OFF), but drive 280 is available (¬STATUS AV) to start DJ:

```
173:  (TAPE-DRIVE-STATUS    ¬RELIABLE? YES    ¬ADDRESS 280    ¬STATUS AV
)
170:  (TAPE-DRIVE-STATUS    ¬RELIABLE? YES    ¬ADDRESS 281    ¬STATUS OF
F)
167:  (TAPE-DRIVE-STATUS    ¬RELIABLE? YES    ¬ADDRESS 282    ¬STATUS OF
F)
```

With this information, JESQ initiates the DJ job so that large jobs on the JES queue can be quickly copied to tape when discovered:

```
53. JM:START-DJ-JOB 5 83 173 60 10
```

The operator receives a message on the expert system console:



Upon initiating DJ, JESQ receives a response from the operating system and updates its internal model:

```
54. JM:DJ-STATUS-UPDATE:FROM-START 8 60 177

179:  (DJ-JOB     ¬STATUS ACTIVE)
```

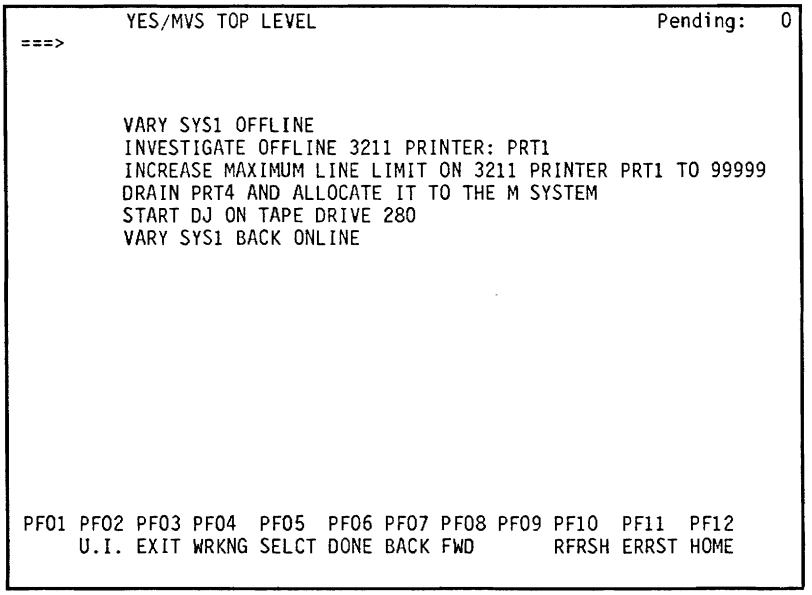
To illustrate another capability, assume that the operator himself submits a JES queue space query to see how JESQ is doing. JESQ will also capture this response in an attempt to use the most recent information. Referring to the trace below, it seems that raising the 3211 line limit has freed a few percentage points of JES queue space, so that JESQ can now operate in solve mode:

```
55. JM:JES-Q-SPACE-UPDATE 8 181 81
56. JM:SET-TO-SOLVE-MODE-FROM-PANIC 7 83 184

184:  (JES-Q-SPACE     ¬PERCENT-LEFT 8)
```

Since 8% space left is sufficient to resume job initiation, JESQ varies the main processor back online:

```
57. JM:VARY-SYS1-BACK-ONLINE 6 186 90 87 10
```

And the operator receives a message on the expert system console:



The operating system confirms the action, and JESQ uses this response to update its internal model accordingly:

```
58. JM:SYS1-STATUS-UPDATE:FROM-VARY 8 90 188

190:  (SYS1    ¬STATUS ONLINE)
```

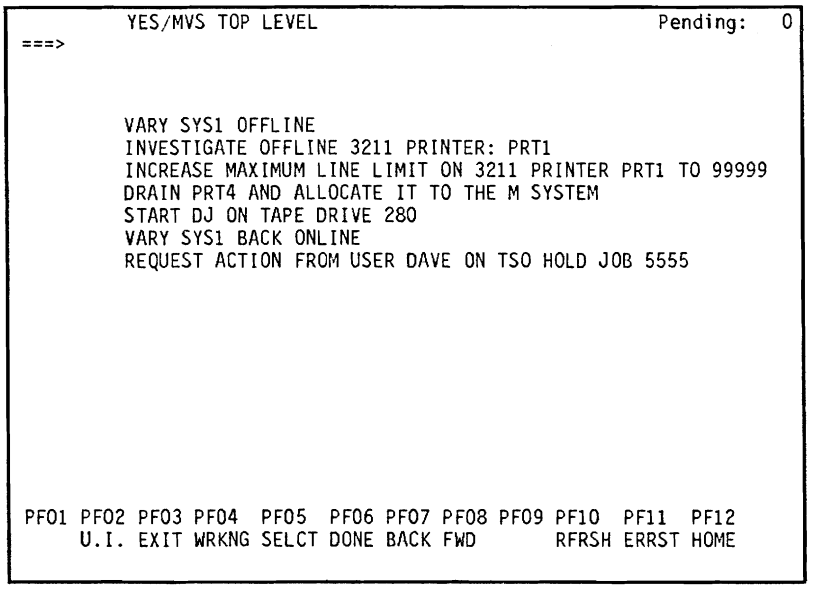
And now the culprit emerges. The response to the query of the datasets held for online viewing arrives, and one such dataset is responsible for problem. An inconsiderate user is storing a million-line dump on the JES queue:

```
192:  (HOLD-QUEUE-STATUS-REPLY    ¬MESSAGE-ID IAT8131    ¬USER DAVE    ¬
JOB-ID 5555    ¬PRINT-LINES 1000000)
```

Were JESQ still operating in panic mode, this dump would immediately be copied to tape (recall that DJ has already been initiated). However, in solve mode, JESQ sends a message to the online user requesting the removal of the dataset and notifying him that failure to do so within ten minutes will result in the removal of his dump:

```
59. JM:REQUEST-USER-ACTION-ON-HOLD-QUEUE-JOBS 5 186 179 192 10
```

Again, the operator receives a message on the expert system console:



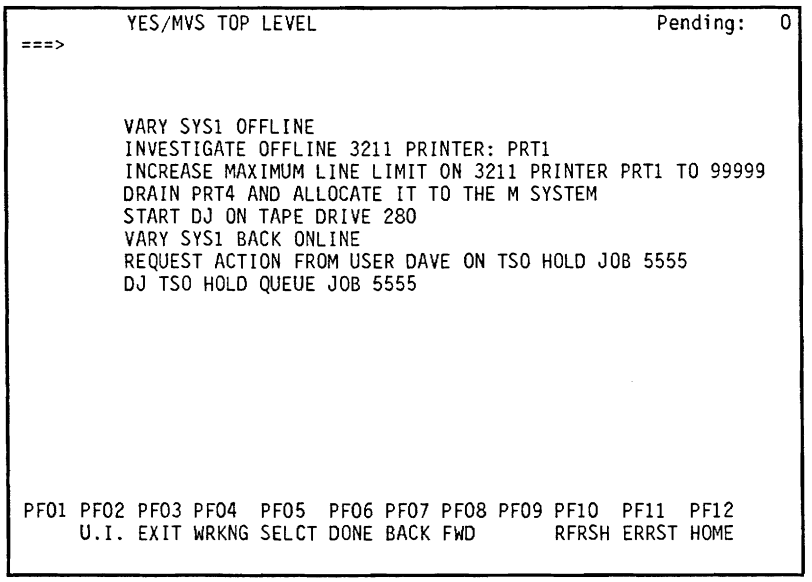
In the meanwhile, JESQ submits another batch of queries and performs model maintenance actions:

```
60. JM:TAPE-DRIVE-STATUS-INVALIDATION:DJ-ACTIVE 3 179 170
61. JM:TAPE-DRIVE-STATUS-INVALIDATION:DJ-ACTIVE 3 179 167
62. JM:SEND-HOLD-QUEUE-STATUS-QUERY 2 186 24 202 10
HOLD-QUEUE-STATUS-QUERY
63. JM:SEND-PRINTER-QUEUE-QUERY 2 186 23 201 10
PRINTER-QUEUE-QUERY PRT4
64. JM:SEND-PRINTER-QUEUE-QUERY 2 186 23 200 10
PRINTER-QUEUE-QUERY PRT1
65. JM:SEND-PRINTER-STATUS-QUERY 2 22 196 10
PRINTER-STATUS-QUERY PRT1
66. JM:SEND-JES-SPACE-QUERY 2 94 195 10
JES-QUEUE-SPACE-QUERY
67. JM:SEND-PRINTER-STATUS-QUERY 2 22 194 10
PRINTER-STATUS-QUERY PRT4
```

Upon passage of the 10 minutes allotted to the user and receipt of the last online job queue response, JESQ notes that the dump is still on the JES queue, and copies the dump to tape, removing it from the queue:

```
214: (HOLD-QUEUE-STATUS-REPLY ¬MESSAGE-ID IAT8131 ¬USER DAVE ¬
JOB-ID 5555 ¬PRINT-LINES 1000000)

68. JM:DJ-HOLD-QUEUE-JOBS-AFTER-USER-WARNING 6 186 179 214 203 10
```

Again, the operator receives a message on the expert system console:



The response to the latest JES queue space query arrives, indicating that the problem has been solved.

JESQ returns to normal mode processing:

```
69. JM:JES-Q-SPACE-UPDATE 8 218 184
70. JM:SET-TO-NORMAL-MODE-FROM-OTHER 7 186 221

221:  (JES-Q-SPACE     ¬PERCENT-LEFT 72)
```

Recall, however, that JESQ has altered a number of parameters in both the external environment and its own internal memory. Upon recognition that queue space is back to normal and of the goals created at the time of corrective action, JESQ resets the appropriate parameters.

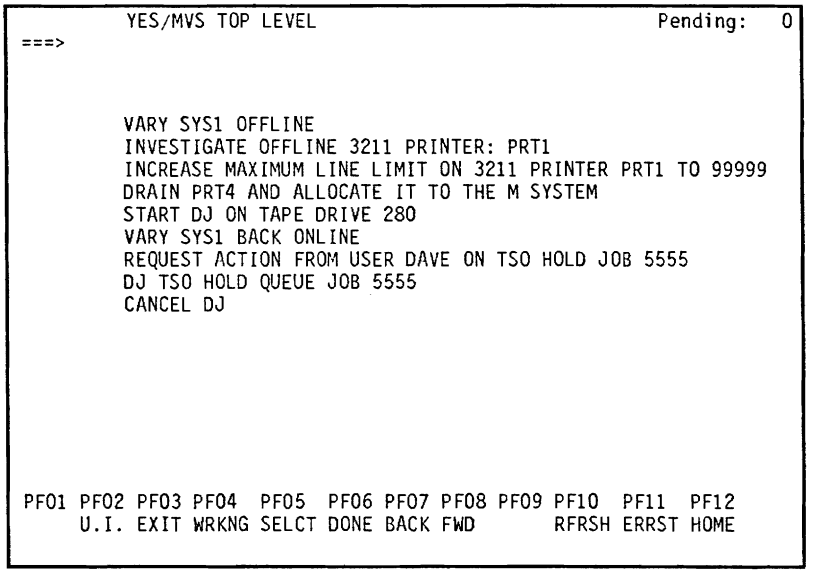
SYS1 was varied offline, but had already been varied back online when solve mode was achieved. However, JESQ speeded up its own queries of the status of the JES queue, and now resets the query interval:

```
71. JM:SET-NORMAL-JES-QUERY-INTERVAL 5 224 94
```

Similarly, JESQ terminates DJ, freeing a tape drive:

```
72. JM:TERMINATE-DJ-JOB 4 223 179 176 10
```

And the operator receives a message on the expert system console:



The operating system confirms that DJ has been cancelled, and JESQ updates its internal model to reflect this:

```
73. JM:DJ-STATUS-UPDATE:FROM-TERMINATE 8 179 229
```

DJ is intended as a facility for temporarily storing JES queue datasets, and the owning users may wish their datasets to be restored. Of course, JESQ could automatically restore the datasets from tape to the JES queue, but this could potentially result in an infinite loop, repeatedly moving the datasets from the queue to tape and back. The operator frequently calls the user once the problem has been resolved to investigate the importance of the dataset, and copies it back to the queue if appropriate. Since JESQ may copy jobs to tape automatically (in "authorized mode") it must additionally list such jobs for the operator so that he can investigate the situation. Thus, JESQ displays the names of the jobs copied (in this case only one) upon recognition that queue space is normal and that DJ is no longer running:

```
74. JM:DISPLAY-DJED-JOBS 5 223 231 217 10
```

And the operator receives a message on the expert system console:

```
         YES/MVS TOP LEVEL                                  Pending:   0
===>


         VARY SYS1 OFFLINE
         INVESTIGATE OFFLINE 3211 PRINTER: PRT1
         INCREASE MAXIMUM LINE LIMIT ON 3211 PRINTER PRT1 TO 99999
         DRAIN PRT4 AND ALLOCATE IT TO THE M SYSTEM
         START DJ ON TAPE DRIVE 280
         VARY SYS1 BACK ONLINE
         REQUEST ACTION FROM USER DAVE ON TSO HOLD JOB 5555
         DJ TSO HOLD QUEUE JOB 5555
         CANCEL DJ
         YOU MAY WANT TO BRING BACK PREVIOUSLY DJED JOB: 5555




PF01 PF02 PF03 PF04  PF05  PF06 PF07 PF08 PF09 PF10  PF11  PF12
     U.I. EXIT WRKNG SELCT DONE BACK FWD       RFRSH ERRST HOME
```

Next, the response to the previous query of the status of the 3211 printer arrives. JESQ recognizes that it has reset the maximum line limit on this printer and that queue space has returned to normal. The internal model is updated upon receipt of the printer status response, triggering the rule that resets the line limit:

```
75. JM:PRINTER-STATUS-UPDATE 8 234 152
76. JM:RESET-3211-MAXLINES 4 223 155 237 10
```

And the operator receives a message on the expert system console:

```
         YES/MVS TOP LEVEL                                  Pending:   0
===>


         VARY SYS1 OFFLINE
         INVESTIGATE OFFLINE 3211 PRINTER: PRT1
         INCREASE MAXIMUM LINE LIMIT ON 3211 PRINTER PRT1 TO 99999
         DRAIN PRT4 AND ALLOCATE IT TO THE M SYSTEM
         START DJ ON TAPE DRIVE 280
         VARY SYS1 BACK ONLINE
         REQUEST ACTION FROM USER DAVE ON TSO HOLD JOB 5555
         DJ TSO HOLD QUEUE JOB 5555
         CANCEL DJ
         YOU MAY WANT TO BRING BACK PREVIOUSLY DJED JOB: 5555
         RESET MAXIMUM LINE LIMIT ON 3211 PRINTER PRT1 TO 30000




PF01 PF02 PF03 PF04  PF05  PF06 PF07 PF08 PF09 PF10  PF11  PF12
     U.I. EXIT WRKNG SELCT DONE BACK FWD       RFRSH ERRST HOME
```

With the queue space problem resolved, and environmental parameters reset to their original values, JESQ waits...

```
77. JM:WAIT 71
```

... and returns to its normal monitoring cycle:

```
78. JM:SEND-JES-SPACE-QUERY 2 227 243 10
JES-QUEUE-SPACE-QUERY
```

This trace demonstrates JESQ's ability to:

- protect the target system from disaster (e.g., vary the main processor offline),
- setup for emergency processing (e.g., start DJ, allocate the 3800 printer),
- keep records for the operator not maintained by the target system (e.g., list of jobs copied to tape),
- buy time in solving a problem (e.g., increasing the 3211 maximum line limit),
- take corrective actions that reflect concern for user satisfaction (e.g., giving the user ten minutes to take action before removing his dataset), and
- alert the operator to problems involving state variables beyond its sensing resources (e.g., the offline 3211 printer).

3.6 THE EXPERT SYSTEM IN ITS ENVIRONMENT

This section briefly describes the expert system's position within the overall configuration of the operations environment as shown in figure 14.

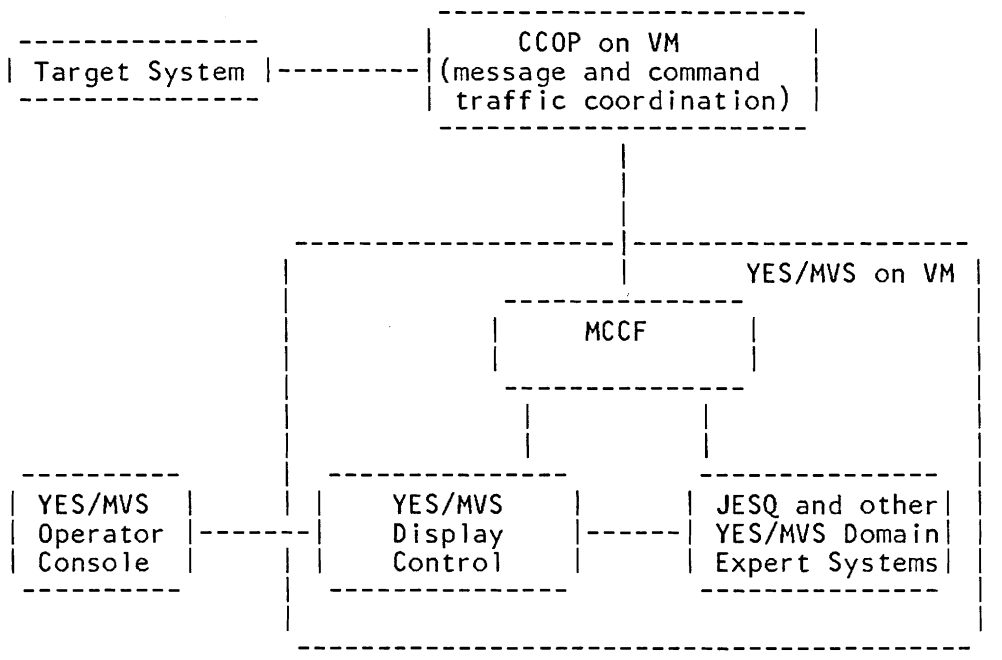


Figure 14: Expert Systems in Operations Environment

In order to insulate the expert system from problems in the target system that it controls, the expert system is run on a separate processor. The requirement that operational expertise be available in the event of target system problems is among the primary factors that discourage implementation of such expertise in the target operating system itself.

YES/MVS runs in three virtual machines under IBM's VM/SP operating system:

1. *The MVS Communications Control Facility (MCCF) Virtual Machine* provides the communications interface between the expert system virtual machine and the target MVS system. MCCF controls the receipt of messages from MVS and the formatting of commands specified by the expert system. IBM Yorktown's Centralized Computer Operation Project (CCOP) [21] is used to capture message streams and to submit commands. MCCF effectively insulates the expert system from the format of MVS and JES messages/commands, passing/receiving the equivalent information to/from the expert virtual machine in OPS5 wme format. This alleviates the knowledge engineer from concern about parsing messages and extracting internal character strings, insulates the expert system from changing command formats as new operating system

releases are installed, and makes the expert system code more readable. Message parsing and command building are accomplished via a table-driven match and translate capability in MCCF.

2. *The Operator Display Virtual Machine* provides the communications interface between the human operator and the expert system machine. This similarly relieves the expert system machine of parsing operator input, formatting display screens, and examining tables that indicate whether commands are authorized for automatic submission to the target system or may only be advised.

3. *The Expert System Virtual Machine* executes the rules in the knowledge base, receiving messages and submitting commands via the MCCF virtual machine. The expert system also sends text to and receives responses from the operator via the Operator Display machine. All of the logically distinct expert systems mentioned in section 1 are executed in this single virtual machine.

This separation of functions provides for the self-containment of the knowledge base, so that operational policies (i.e., rules) are more easily read and modified.

4. JESQ DEVELOPMENT METHODOLOGY

The development of JESQ is described here to augment the data on the development of expert systems in industrial environments. While literature that describes such projects (e.g., [22]) is beginning to emerge, it is currently sparse. The development of JESQ is interesting in this regard from both organizational and technical points of view:

- *Organizational Challenges*: YES/MVS as a whole represents an obvious threat to operations staff. Yet the knowledge engineering process requires significant time, effort, patience, and interest on the part of the experts involved. While the perceived threat from expert systems represents a potential problem for any knowledge engineering effort, this is more of an obstacle in the problem domains of non-unionized non-professionals. Unlike the real-world domains to which expert systems have generally been applied, automated operations poses a realistic danger to those employed in computer operations, especially the operators themselves.

- *Technical Challenges*: YES/MVS is among the first expert systems that executes continuously in realtime and exerts active control over the environment it monitors. The implications of these characteristics go beyond the architectural issues discussed thus far, affecting testing methodology as well.

The following sections describe the development of JESQ in detail.

4.1 KNOWLEDGE ACQUISITION

The primary knowledge sources for JESQ included an operations expert dedicated full-time to the YES/MVS project, MVS operators and operations managers, resident systems programmers, system manuals, and operator console traces of responses to actual problems.

The bulk of the JESQ knowledge base was derived from informal interviews with the primary expert and operations staff. After first studying MVS manuals to learn basic concepts and vocabulary, the

author held a series of long and intensive meetings with the primary expert, culminating in the formulation of a conceptual framework for managing JES queue space.

The author then approached several operators on different shifts individually, gaining additional space management strategies and reviewing the strategies already collected. Systems programmers were consulted in cases where operators could not offer a justification for their actions beyond their specification in installation procedures. Where operators offered conflicting viewpoints, operations managers (possessing a more global perspective) provided additional clarification.

Next, the author produced a functional document detailing a comprehensive strategy for managing JES queue space. This document was distributed to the knowledge sources consulted, requesting feedback regarding omissions and misinterpretations. This served the dual purpose of correcting the author's misconceptions while generating interest among the operations staff by further involving each of them in the project. Gathering the feedback of several operators also served to uncover considerations and additional strategies that had not surfaced in previous discussions. In general, operators seemed more interested in the notion that their personal problem resolution strategies would potentially be implemented throughout IBM than fearful that their skills would be devalued by automation. In part, this was due to their confidence in IBM's policy of retraining employees whose skills have become obsolete.

A period of intensive observation followed. The author spent several nights on the "bridge" watching the operators perform their tasks in a real setting and examined traces of operator actions when incidents occurred. These activities, too, unearthed several details that the experts had omitted or not emphasized. A common theme in expert systems development is that experts often omit or only briefly discuss knowledge that is entirely opaque to the knowledge engineer. In part, this is what motivates an architecture that facilitates incremental development.

A prototype system was developed to verify that the spirit of the problem had been captured. Following verification of the prototype by the experts, a long period of iterative information gathering, testing, and debugging commenced. Information gathering took the form of additional discussions

and "rule walkthrus" with the operations staff. Of course, testing against the live target system under operator supervision provided the richest environment for knowledge refinement.

4.2 TESTING AND KNOWLEDGE BASE REFINEMENT

A combination of techniques were employed to test JESQ. The ideal would have been to construct an MVS simulator against which to test JESQ, but the complexity and dynamics of the MVS environment prohibited this approach. Thus, it was necessary that the real environment serve as the primary testing ground for JESQ.

Trivial logic bugs and syntax errors were removed in a "timeless" environment by loading JESQ with fabricated target system responses to mock actions. A "random sabotage" scheme was employed in the real environment under which the JES queue was saturated with jobs of various characteristics (each recognized by different rules in the knowledge base) and key devices were disabled. A "controlled sabotage" scheme was also employed, involving the manipulation of specific resources to trigger specific rules that would not normally be invoked in the majority of problem situations.

Testing in the real environment presented several challenges. Communication difficulties presented a problem in that target MVS system messages were sometimes lost in transmission. Of course, a human operator experiences the same problem.

Testing costs presented another difficulty. For example, JESQ may trigger the printing of a large mock dump during testing. While this action would be appropriate in an actual crisis, the computing center was naturally reluctant to absorb the cost of printing such a dump for testing purposes.

Balancing testing objectives with service degradation introduced another testing consideration. While tests involving gross sabotage of the target system were performed off shift, some number of users were nonetheless affected.

Reminiscent of operating system testing, it was often difficult to duplicate the conditions under which a bug had previously been uncovered in order to test its subsequent correction. Recall, however, that

portions of the target system are explicitly represented in JESQ (in working memory), as are the conditions that trigger JESQ action (in the antecedents of the rules). Thus, some of these conditions were easily identified. Of course, the difficulty arose in those cases where an action repeatedly failed due to target system behavior that had not been explicitly modelled in JESQ.

Another problem concerned the ability to distinguish between operator objections to JESQ actions that could be attributed to style rather than substance. Again, one of the motivations behind JESQ is to standardize operator activities and make this style-substance distinction once and for all, leaving it recorded in the knowledge base. To the extent that individual style clouds the evaluation of JESQ's performance, it may be viewed as a testing difficulty. Of course, separating style from substance is general problem for knowledge engineering and is not particular to JESQ.

A significant problem concerned testing an automated facility in an "advisory" mode. As described, the expert system can (as controlled by software switches that may be reset by the operator) either take actions automatically or can give advice to the operator on how to handle situations. However, there exist significant differences between fully automated execution and advisory execution from the expert system's perspective. For example, handling high volumes of queries, responses, and actions are among the principle advantages of automation, yet testing in advisory mode discourages the output of high volumes of advice. Also, the semantics of ignored advice are ambiguous, failing to distinguish between erroneous advice and a busy operator. A related issue, since JESQ may offer recommendations faster than a human operator can execute them, such recommendations may expire by the time the operator carries them out. Multiple recommendations regarding the same entity may accumulate on the operator's screen over time, creating confusion regarding which advice is appropriate.

While testing a realtime active expert system has not been trivial, the author has tested most actions encoded in the knowledge base and has produced the desired behavior as defined by the operators observing the tests. Of course, such informal testing lacks the rigor of a formal evaluation, which is discussed in the next section.

4.3 FORMAL EVALUATION

JESQ runs regularly at the IBM Yorktown Computing Center and has received a favorable response from the operations staff. The system runs in advisory mode during the day, and fully authorized at night. While not enough data has been collected under experimental conditions to report the results of a formal evaluation, the requirements for such an evaluation are outlined in this section.

A formal evaluation of JESQ must measure the criteria of interest to each of the affected groups at the installation. These criteria vary by group as follows:

- *Operations Management* will be most interested in the improvement in target system availability and performance resulting from automating queue space management. The flexibility with which installation policy is reviewed and modified is also of primary importance to this group. In addition, management will be concerned with the availability and performance of the expert system itself.

- *Operations Personnel* will focus on the performance of the expert system at the action level. Parameters to measure include the correctness and timeliness of space management actions, and their sequence of execution. Operators will also be concerned with the human engineering aspects of JESQ, particularly its ability to clearly identify the context in which actions have been performed or advised.

- *System Programmers* will be concerned with the understandability and modifiability of the JESQ knowledge base and their ability to test it. Regarding their role as the correctors of system problems, systems programmers will also be concerned with JESQ's consistency of action, and incident report uniformity and content.

- *Performance Analysts and Capacity Planners* will be similarly concerned with consistency of action, and incident reporting uniformity and content.

Ideally, this data would be collected from several sites running JESQ, channeled to a central collection point, and collectively analyzed.

5. CRITICAL REVIEW AND ALTERNATIVE APPROACHES

The current implementation of JESQ successfully resolves JES queue space problems under test conditions in the real environment, and should provide the simplicity of representation required for continued review and maintenance of the knowledge base. However, suboptimalities in JESQ can be identified. Some of these are particular to the application while others more reflect the limitations of current expert systems technology. An examination of the problems in JESQ in light of the requirements for ongoing maintenance, and the identification of generic types of knowledge inherent in realtime active process monitoring tasks together suggest the development of a language for process monitoring expert systems.

5.1 LIMITATIONS OF JESQ

This section describes a set of related shortcomings in JESQ, focussing on issues of program control and knowledge representation, and their suitability in the context of the requirements of the application.

Granularity of Knowledge Representation

JESQ is intended to approximate the level of description found in an installation run book. JESQ's obvious limitation in this regard is that it encodes not only high level policy descriptions, but specifications for query submission, response collection, and data manipulation as well, all in the same representational framework (production rules). Most run books are not written at this level of detail, but knowledge of when to solicit information, what information to solicit, and how to solicit that information must be encoded *somewhere* in JESQ.

Of course, once the information gathering structure has been built (i.e., rules have been coded to solicit, collect, and maintain the required information), natural looking policy rules can be added to and

deleted from the Knowledge-Based Action rule group without modifying other portions of the knowledge base in most cases. In those other cases where a policy that relies on information not already maintained by existing rules must be added to the Knowledge-Based Action group, JESQ's rule group classification scheme facilitates the identification of the rules needed to support the newly-added policy (i.e., every condition is supported by a query submission rule, a response collection rule, etc.). However, specifying supporting processes at the same level of representation as high-level policies can be viewed as something of a limitation in that they really represent two different kinds of knowledge. Declarative rules are most naturally employed in JESQ to encode knowledge of *what to do* under a specified set of conditions, whereas descriptions of processes that support the verification of such conditions and of the steps by which the associated action is performed more express knowledge of *how to do* something. For example, the policy rule of Figure 4 (section 3.2) in some sense expresses *what to do*: IF there exists a queue space problem AND there exists a job in hold status which exceeds 30000 lines THEN release the job. *How to do* this may be expressed as: submit queries regarding the status of JES queue space and of the job queue and collect the associated responses, update JESQ's internal model accordingly, and finally, submit the command that will result in the release of the job, provided that the mentioned conditions are satisfied.

JESQ could be improved in this regard by providing programming constructs which allow for the separation of knowledge about *what* from knowledge about *how*. One way to accomplish this separation would be to introduce the notion of *action procedures* which specify the submission of queries and collection (and manipulation) of responses required to verify the conditions under which a potential action is appropriate, and the sequence of steps required to accomplish that action. Such action procedures could then be embedded in a planner that schedules potential actions on the basis of their anticipated utility in solving the problem at hand.

Eligibility Knowledge, Modularity, and Local Intelligence

Again, our premise is that we want the installation to think of JESQ (with regard to execution and maintenance) as an automated run book, rather than a computer program. While JESQ is designed with the objective of mapping operational policies into isolated sets of rules, its maintainers will still, however, be unable to disregard considerations of program control. That is, JESQ is a computer program, not a run book, although its architecture is motivated by an attempt to blur this distinction, and does so with a reasonable degree of success.

JESQ's knowledge base is not devoid of control constructs, in part, because the knowledge-application heuristics built into the OPS5 inference engine do not closely mirror those of human operators who "apply knowledge" in the JES queue space domain. A general expert system building tool, OPS5 strives to implement general (domain-independent) problem-solving heuristics, omitting an isolated structure in which to store what we might call *action eligibility knowledge* in the context of process control applications. For example, knowledge that an action should only be attempted *n* times over a particular temporal interval would be classified as eligibility knowledge. There is no place to store such knowledge in OPS5 other than in the LHS of rules, and we would like to encode only descriptions of operational events in the LHS, since the recognition of such events and their associated actions are what the experts consider to be their expertise. Eligibility knowledge is regarded more as "common sense" knowledge in this domain. For example, it would not occur to an experienced operator to explain to a novice that it would be inappropriate to attempt a given action every few microseconds. Creating a separate location in JESQ to house eligibility knowledge would serve to illuminate the presence of such knowledge for knowledge base maintainers, while allowing the encoding of event-response pairs that omit such eligibility knowledge from their LHS.

Thus, the problem in this area is that selecting instantiations (i.e., rules and the wmes that match their LHS) on the basis of recency of information and specificity of conditions is of limited value in the JESQ domain, omitting criteria like "absolute successful execution limit" and "minimum intervals between attempted executions" for selecting potential operational actions. Augmenting the algorithm

with the capability to resolve rule conflicts across functional groups (see section 3.3) results in a knowledge application strategy that more closely conforms to the requirements of action selection in the queue space management domain (implementing selection heuristics such as "delete old responses before submitting queries that will generate fresh ones"), but is still too general to allow installation policy makers to add event-response pairs to JESQ without considering their effects in the global context of the system's execution. That is, where the implemented conflict resolution criteria fail to capture knowledge-application requirements, control mechanisms must be coded in the rules themselves. In part, the goal of knowledge modularity and independence is achieved to the degree that such control mechanisms are minimized. That is, we want the ability to add policies (event-response pair rules) in a *local* fashion, and explicitly specify knowledge about when such policies are to be invoked to the interpreter in such a way that this knowledge is isolated from the event-response pairs that are naturally expressed in production rules. In lieu of this capability, the knowledge engineer is forced to achieve a balance between (1) modularity and uniformity of knowledge representation and (2) performance degradation due to the omission of explicit control knowledge in the rules.

The cost of the modularity achieved in JESQ is a certain "local intelligence". The decoupling of query submission, response collection, garbage collection, and knowledge-based command submission, coupled with the omission of action eligibility knowledge in JESQ results in suboptimal (though acceptable) performance in some cases. For example, the firing of a rule that submits an effective corrective command may be delayed by the asynchronous submission of queries that ultimately supply the state variables required to trigger that rule. In the spectral case, a rule that submits a corrective command may be fired "too many" times over a temporal interval due to the synchronous arrival of multiple response wmes which trigger that rule. Of course, these problems could be eliminated by encoding absolute firing limits or firing limits over temporal intervals in the rules themselves, but, again, we want to isolate such "common sense" knowledge from the event-response pairs that represent operational policies.

These behaviors should not be viewed as *problems* for JESQ, but suboptimalities. In the case of delayed rule firings, the delays are short since queries are submitted often. Multiple rule firings as described only occur on occasion, and there is virtually no cost to submitting a corrective command to the target system that ultimately fails (perhaps due to the effects of the last command submitted) above that of the additional query traffic and operating system processing. In fact, the experts (operators) exhibit both these behaviors. On the other hand, this architecture actually promotes JESQ's performance under some circumstances, since the system does not wait to complete an action before attempting another, potentially resulting in the completion of several actions over a shorter interval of time than would be possible if query submission, response collection, and command submission were serially scheduled. Nonetheless, this scheme does sacrifice understandability of behavior, and economy of communications and operating system resources for simplicity and uniformity of representation.

Global Planning and Focus of Attention

In general, actions are opportunistically invoked outside the context of specific subgoals in the current implementation of JESQ. JESQ's model of the world largely consists of snapshots of resource states, omitting historical data regarding its recent actions (goals for reset actions being the exception). Policy rules are executed whenever an anticipated event is recognized by the match process as indicated by such snapshots.

This "flat" architecture is suboptimal, in that a shift of focus is often desirable over the course of the problem solving process. For example, if the operator is unable to solve a queue space problem over some temporal interval, it may be appropriate for him/her to focus (for a short time) on actions that may potentially increase queue space by just a few percentage points in an effort to buy time before resuming the search for the primary causes of the problem at hand. Of course, rules that buy time in problem solving are included in the current implementation, but are not explicitly identified as such. Consequently, the current implementation cannot explicitly reason about its own focus. Rather, rules

are invoked whenever their conditions are satisfied, be they "time-buying" rules, "problem-finding" rules, or rules that reflect other high level goals.

An architecture which better reflects operator behavior would encode a more hierarchical view of the problem solving process in the queue space management domain, incorporating a goal-directed planning scheme that is driven by the recent history of JESQ's actions. This would probably improve JESQ's performance while rendering JESQ's behavior over the course of the problem resolution process more intelligible to operations staff.

Calling for Outside Help

The undirected firing of rules robs JESQ of the ability to call for outside help, a common shortcoming of expert systems. Since JESQ's internal model of the target system is a collection of snapshots rather than histories, JESQ does not encode the notion that it has tried every trick in its knowledge base at least once without success. The capability to call for outside help would serve the dual purpose of instructing the operator to consult an external source (e.g., systems programmer) at the appropriate time, while helping to identify omitted knowledge in cases where the operator is actually able to solve the problem himself after JESQ has "given up". A representation is needed that allows JESQ to behave is if it knows when it does not know how to solve a problem.

Desirability Knowledge and Depth of Reasoning

JESQ employs the added priority conflict resolution mechanism (see sections 3.2 and 3.3) to resolve conflicts between multiple policies (Knowledge-Based Action rules) that are simultaneously applicable (as well as to resolve conflicts across functional rule groups). One of three priority classes (low, medium, high) is associated with each policy rule. Once these priorities have been coded, however, their underlying justification is lost, and this, to some extent, impedes both performance and effective

knowledge base maintenance. The generic factors (in the JESQ domain) that underlay the selection of an action's priority include:

- anticipated impact on queue space resulting from successful execution of the action;
- how convenient the action is for the human operator to perform;
- how resource intensive the action is to perform;
- the anticipated level of user satisfaction resulting from successful execution of the action;
- the "track record" of the action over the cumulative history of its execution;
- the anticipated speed with which the action is executed by the operating system or human operator;
- the general availability of the resources required (conditions under which the action is appropriate) to perform the action;
- the probability of acquiring tne necessary resources (setting up the necessary conditions) if they are found to be unavailable; and
- the "recent" success of the action.

Formulating priorities for new Knowledge-Based Action rules is non-trivial, for these factors are not explicitly represented in JESQ, and future maintainers will not be forced to consider them, nor are any two maintainers likely to combine these factors in exactly the same way to arrive at the appropriate priorities. Moreover, the priority scheme places too much computational importance on a single heuristically justified number. In effect, the semantics of all the above-listed factors have been bundled into a single symbol.

What is needed, then, is a representation that allows us to retain the objective of encoding surface behavior in event-response pairs, while also allowing the representation and employment of the factors that underlay the selection of event-response pairs when more than one are applicable. That is,

we want the capability to encode and use *desirability knowledge* in resolving conflicts among competing actions that could potentially satisfy the same goal. Desirability knowledge is different from eligibility knowledge. While eligibility knowledge is used to disqualify potential actions on the basis of recent history, desirability knowledge is used to select among competing actions that have already been deemed eligible. Informally, eligible means "might be a good thing to try now" where desirability means "might be the best thing to try of those that might be good to try now".

Again, to keep this discussion in perspective, the potentially suboptimal assignment of priorities to potential actions in the current implementation is not really a problem in a practical sense, since JESQ continuously "recovers" from its previous behaviors by attempting different actions as opportunities arise. For example, if the appropriate priorities for two actions are mistakenly reversed, the less important action may be initiated a few seconds before the more important one. In most cases, this represents a trivial flaw in JESQ's performance.

Summary

Several problems of representation and control in JESQ were identified, although these do not represent practical concerns in light of JESQ's performance requirements. These problems are largely due to the inability to naturally encode different generic types of knowledge in the chosen "flat representation", and to failure to embed the operational policy descriptions in a more global planning framework. In the next section, an alternative architecture is outlined which could potentially eliminate many of these problems. The hope is that these ideas can be extended to support realtime process monitoring tasks in general, leading to the development of a general expert systems language for realtime process control tasks.

5.2 A THEORY OF JES QUEUE SPACE MANAGEMENT

Ideally, JESQ would implement a *theory* of queue space management, circumscribing the task within a framework that abstractly captures its generic components and forces the installation to conceptualize and communicate policies to JESQ in terms of those components. That is, JESQ should supply distinct knowledge stores for the different types of knowledge identified in section 5.1, and provide for the declarative specification of this knowledge, free from programming considerations. In this ideal world, management would truly be providing JESQ with policy *data* rather than the declarative *programming constructs* (rules) that approximate such data. That is, given that the implemented theory fully parameterized all potential policies, installation management would come to *think in JESQ* in writing new policies, much in the way that PASCAL programmers *think in PASCAL* in writing new programs. The crucial difference, of course, is that "thinking in PASCAL" involves pondering loops, variables, and other programming constructs, whereas we would like "thinking in JESQ" to involve pondering queries, actions, and space freeing strategies, free from programming considerations. In this context, a weakness of many current expert system shells is that knowledge base maintenance involves thinking about both the domain concepts represented by programming constructs *and* their computational role in the chosen programming paradigm.

The architecture of the "queue space theory system" would include an inference engine that reflects the heuristics of action selection in the queue space management domain, and internal model maintenance facilities that are tightly coupled with the actions they support. Policy descriptions would be linked to high level goals, and competing actions explicitly identified within this context. Such an architecture is outlined below, although no part of it has been implemented and no claims are made as to its completeness. This scheme might prove to be generally applicable to monitoring task domains, although this claim cannot be supported in lieu of implementing the scheme in several such domains. The architecture is outlined here primarily to clarify the "ideal" put forth, to show how some of the weaknesses outlined in the previous section might be eliminated, and to lay the foundation for further work.

Representation

In the hypothetical framework, event-response pairs are embedded in *action-procedures* along with their supporting processes. An action-procedure consists of primitive actions such as:

- query-the-status-of-environmental-resource-*x*,
- collect-and-analyze-query-response-set-*x*, and
- submit-action-command-*x*

where *x* depends on the action-procedure at hand. The mission of an action-procedure is to determine if the conditions under which an action is appropriate are satisfied, and, if so, to execute the primitive actions that comprise the action intended.

Action-procedures are, in turn, grouped in terms of domain-dependent goals. In the queue space management domain, these include:

- *Protect remaining space*: For example, vary the main processor offline, preventing additional queue space depletion.
- *Remove space clearing barriers*: For example, reset parameters on printers to print special forms that are backed up on the queue.
- *Remove large jobs*: Search for and either cancel, print, direct user attention to, or DJ large jobs.
- *Buy time*: Print or otherwise clear smaller jobs from the queue periodically while searching for the real problem.
- *Set up for problem resolution*: Perform setup actions for potential space clearing actions. For example, start DJ and raise the line limits on printers.

- *Restore environment*: Reset environmental parameters that were manipulated during the problem solving process. For example, terminate DJ and reset the line limits on printers.

- *Call for help*: Notify the operator that JESQ has exhausted its knowledge base at least once and request his/her help.

- *Monitor*: Measure the space remaining on the JES queue and reconsider the appropriateness of the current goal.

Thus, goals abstractly represent groups of competing event-response pairs, and such pairs are embedded in procedures which submit the required queries, collect the associated responses, and submit the appropriate action command(s), provided that action preconditions are satisfied.

Control

Goals are instantiated by rules that examine the recent history of JESQ's activity at the goal level. For example, the rules of Figure 15 might be included in the goal instantiation rule set:

```
IF   processing-mode = panic;
and  remove-space-clearing-barriers has been executed 3 or more times
        over the last 20 minutes;
THEN instantiate buy-time

IF   processing-mode = panic;
and  remove-space-clearing-barriers has been executed 3 or more times
        over the last 30 minutes;
and  buy-time has been executed over the last 10 minutes;
THEN instantiate call-for-help.
```

Figure 15: Goal Instantiation Rules

The current goal dictates the action-procedures that can potentially be instantiated and executed (although action-procedures can span multiple goals). An action-procedure linked to the current goal is considered to be a candidate for instantiation if it is *eligible*. Attributes that can render an action-procedure ineligible include: the action-procedure has already reached its absolute-successful-execution-limit or execution-limit-over-a-temporal-interval, or the minimum-intervals-between-attempted-executions has not yet passed. Action eligibility is expressed solely in terms of problem severity and triggering conditions in the current implementation of JESQ.

In the hypothetical implementation, triggering conditions are not confirmed (i.e., queries are not submitted) until an action-procedure is selected as eligible and desirable on the basis of its context in recent history and its relationship to goals.

Eligible action-procedures are instantiated on the basis of *desirability*. In the current implementation of JESQ, desirability is implemented by conflict set reduction by priority and by the OPS5 conflict resolution algorithm. That is, competing policies with satisfied conditions are resolved in accordance with the priority attached to the policy, the recency of the information that drives the eligibility of the action, and the number of conditions that need be satisfied to render the action eligible. In the hypothetical framework, action desirability is expressed in terms of the semantics of "desirability" in the queue space management domain (as listed in the previous section). The factors underlaying action-procedure selection are explicitly represented and assigned values at a coarse enough grain size (say, LOW, MEDIUM, and HIGH), to approximate an isomorphism between their meaning to an operations policy maker and their role in the algorithm used to compute desirability. The objective here is to provide few enough choices for values so that such values can be assigned locally (i.e., without comparing with other values in making the assignment).

Desirability is not computed numerically, but rather, symbolically, again, in terms meaningful to the administrator. The basic idea may be informally stated as follows: A common set of factors underlays the selection of all actions. Depending on the severity of the problem at hand, these factors will vary in relative importance. The most desirable of a set of actions that satisfy the same goal is that action which, depending on the severity of the problem at hand, meets or exceeds the specified requirements for underlying selection factors. If no actions qualify given these requirements, then some requirements are relaxed. The specific factors for which requirements are relaxed, as well as the rate at which they are relaxed, will vary with the severity of the problem at hand.

Under this scheme, a friendly front end maps user responses (LOW, MEDIUM, HIGH) into an *Action Selection Template* for each action, providing values for each of the criteria underlying its selection as shown in figure 16. Some values are updated (learned?) (e.g., track record) with each ex-

ecution, while others remain static as originally coded (e.g., operator convenience). Initial guesses are provided for "learned" factors.

```
ACTION_SELECTION_TEMPLATE
   ACTION-ID: dj-a-job
     anticipated impact                                HIGH
     operator convenience                              LOW
     resource conservation                             LOW
     user satisfaction                                 MEDIUM
     track record                                      HIGH
     speed of action                                   LOW
     general availability of necessary resources       HIGH
     probability of acquiring necessary resources      HIGH
     recent success                                    HIGH
ACTION_SELECTION_TEMPLATE_END;
```

Figure 16: Action Selection Template for dj-a-job

The importance of each of the Action Selection Template criteria varies with the severity of the problem at hand (processing mode). For example, user satisfaction is of prime consideration when space is not at a critical level, but is of less importance in a panic situation. Speed of action is more important in panic mode than in poke mode, etc. For each processing mode, the user similarly defines (via some friendly front end) a sequence of action-independent *Selection Criteria Relaxation Templates* as shown in figure 17.

```
PROCESSING-MODE: panic
   RELAXATION_TEMPLATE 1 of 8
     anticipated impact                                HIGH
     operator convenience                              HIGH
     resource conservation                             HIGH
     user satisfaction                                 HIGH
     track record                                      HIGH
     speed of action                                   HIGH
     general availability of necessary resources       HIGH
     probability of acquiring necessary resources      HIGH
     recent success                                    HIGH
   RELAXATION_TEMPLATE_END;
```

Figure 17: Criteria Relaxation Template #1 for Panic Mode

Each Relaxation Template in the sequence lists each of the factors that underlay the selection of competing actions and its associated lower bound (where LOW < MEDIUM < HIGH) for matching against Action Selection Templates in choosing among actions linked to the current goal. In effect, the full sequence of Relaxation Templates specifies (1) which criteria are to be relaxed in choosing among competing actions and (2) the rate at which criteria are relaxed.

The action selection algorithm attempts to match each template in the Relaxation Sequence against the Action Selection Templates associated with action-procedures linked to the current goal until a match on all lower bounds (or better) is found. Thus, the first Relaxation Template relaxes no criteria: A search for an action (Selection Template) of HIGH anticipated impact, HIGH operator convenience, etc. is performed. If none is found, the next Relaxation Template (presumably specifying LOW or MEDIUM for at least one criterion as a lower bound) is invoked and the attempted match is repeated. The search continues until a match is found. Presumably, the last Template specifies LOW as the lower bound for all criteria, guaranteeing a match.

This scheme not only captures information that would be lost in the current implementation, but additionally relieves the knowledge base maintainer of performing the unstructured task of priority assignment. Moreover, the computational importance of any single encoded symbol is diminished. The intent is that no single criterion be terribly important in the action selection algorithm and that the grain size of the values assigned to selection criteria be coarse enough to have meaning in isolation.

Thus, potential actions are selected on the basis of eligibility under the current goal, and competing actions are resolved on the basis of desirability.

Unlike the production systems implementation under which one action is instantiated and executed at a time (i.e., a rule is fired), the hypothetical system instantiates as many action-procedures as are judged to produce a noticeable effect on the status of queue space and executes the resulting plan. Thus, the relationship between planning and execution under the hypothetical scheme may be informally stated as: "Plan (instantiate action-procedures) under the assumption that all potential actions selected will encompass conditions that may be satisfied by explicit setup actions or will already be satisfied and will successfully execute. Then examine these conditions via sensor readings (queries), attempt to satisfy them if necessary, and take action accordingly. Finally, examine the state of the target resource to see the results and replan from there". (See [23] for a general discussion of the relationship between planning and execution. McDermott's notion is quite different). Corresponding to the last step in this description, the last action-procedure instantiated is always *reassess.*

Reassess triggers a query of the state of JES queue space and returns control to the top-level goal instantiation rules. If the state of the queue is much the same (within predefined thresholds) as the last assessment, the action-procedure instantiation process is performed again using the current goal. A new set of action-procedures will usually be eligible and the previously instantiated set ineligible because of eligibility criteria such as minimum-intervals-between-attempted-executions. Should the state of the JES queue differ from the last assessment, a new goal is instantiated which then drives action-procedure instantiation. If no action-procedures are eligible under the current goal, it is removed and a new one is instantiated (possibly *Call for help*). This does not disrupt the continuity of action eligibility data (e.g., execution-limit-over-a-temporal-interval), because this data is maintained at the action level and is independent of the current goal.

Summary

Summarizing the key features of the hypothetical framework, the control cycle is driven by the continuous examination of recent problem solving history, instantiating new high-level goals when appropriate. The system's intent to execute an action is in accordance with the current goal and explicitly represented constraints over temporal intervals (i.e., eligibility knowledge). Competing actions are resolved on the basis of explicitly represented domain-specific factors (i.e., desirability knowledge). The submission of queries that examine action-specific conditions is limited to potential actions that have already been instantiated on the basis of eligibility and desirability.

This scheme more resembles a "theory of space management" than the current architecture, providing a global framework for selecting and coordinating actions that (to some extent) mirrors the heuristics of action selection in the domain. In effect, the hypothetical framework embeds the event-response pairs of the current architecture in a planner that is driven by domain-specific scheduling heuristics. Since some degree of uniformity is provided across all actions, this framework is also advantageous in that it could be used to support a limited form of automated knowledge acquisition and explanation generation. Explicit representation of the factors that underlay action selection could also improve JESQ's potential usage as a training tool, providing a basis for answering comparative

user queries of the form "Why is action *x* preferable to action *y* in this situation?", with responses of the form "Because user satisfaction is HIGH for action *x* and LOW for action *y*, although operator convenience is ...". Hypothesizing that the implemented theory is complete, *never* requiring modifications, management would merely have to fill in the domain knowledge. That is, the implemented theory would force management to describe policy in terms of the theory, supplying policy data rather than declarative programming constructs that implement policy in the context of a more general interpreter's execution.

Again, this framework is incompletely specified, and problems would undoubtedly surface upon beginning implementation. The framework has been presented here to provide a rough sketch of what we might set out to accomplish in pursuit of the goal of developing a language for encoding policies and associated knowledge for process control tasks, specifically for the queue space management domain. The next section addresses the potential utility of attempting to develop such a language from both the short and long term perspectives, and outlines alternative approaches to its construction.

5.3 ALTERNATIVE APPROACHES AND RESEARCH DIRECTIONS

The preceding section outlined the design of a "theory of queue space management" as an alternative to the current JESQ architecture. The hypothetical framework provides for the natural encoding of the different types of knowledge that comprise queue space management (and hopefully other monitoring tasks), going beyond the flat representation of the production systems architecture. This section addresses the questions of (1) the utility of implementing the described alternative, and (2) how to proceed in doing so.

In response to the first question, we must first recognize the potential costs of prematurely implementing the described "theory" beyond that of the obvious costs of development. The danger here is that omissions in the implemented theory would culminate in the very problems that motivated the construction of such a theory: "Hacks" would creep into the domain knowledge representation to

compensate for omissions in the theory itself. This might not seem so bad at first glance, since this problem characterizes the current implementation. However, under the new architecture we might be worse off than before. Since the hypothetical computational formalism is more complicated than the original architecture, it would become more difficult to "hack" the desired behavior when this behavior could not be achieved via the implemented representation. Thus, in the short term, a straightforward architecture based on a simple, general tool such as OPS5 probably represents the most reasonable approach. The current JESQ architecture satisfactorily performs its intended task, provides for the natural expression of event-response pairs, and yet is based on a simple enough programming formalism to allow one to "hack" the desired behavior in exceptional circumstances.

From a research point of view, however, we want to formalize the problem solving process and produce as general tools as possible. Thus the longer term perspective on this first question suggests the development of a language specifically designed for realtime process control expert systems. Generalizing from the objectives of the "queue space management theory system", the goal is to identify the different types of knowledge that comprise intelligent process control and to develop representations that ultimately allow system users to specify the elements of a particular process in simple declarative statements. This raises the second question put forth, that of how to proceed in building such a system, and there are at least two approaches to consider: (1) We can implement very general theories, attempt to apply them in several process control domains, and expand the theories as new requirements are uncovered; or (2) We can implement more specific theories of particular domains in isolation and then step back to identify the abstract commonalities among them with an eye toward the development of more general systems that subsume these commonalities.

It might be valuable to view process control expert systems as a subclass of expert systems united by common requirements (some of which are described in section 2.7) and proceed in both directions. A preliminary in-depth examination of many potential application domains and of traditional process control applications would be helpful in establishing the appropriate course of investigation.

6. SUMMARY AND CONCLUSIONS

The application of experts systems techniques to process control domains represents a potential approach to managing the increasing complexity and dynamics that characterizes many process control environments. Expert systems may be of potential value as high-level supervisors in current computer-controlled systems that employ rigid, predetermined control sequences, and in domains which currently require human operators to monitor and control some process on a continuous basis. This thesis reported on the application of expert systems techniques to the domain of computer operations.

The requirement for high availability, high performance, computing systems has created a demand for fast, consistent, expert quality response to operational problems. Effective, flexible automation of computer operations would satisfy this demand while improving the productivity of operations. However, the operations environment is characterized by high complexity and continuous change, rendering it difficult to automate operations in traditional procedural software. This situation is exemplified by the domain of JES queue space management, the focus of this thesis, and probably generalizes to many process control domains.

The techniques of expert systems provide a basis for automating operations despite the complexity, the lack of formalization, and the dynamics that characterize the operations environment. In particular, the modularity of knowledge encoding encouraged by the production systems architecture provides for the incremental development, ongoing modification, and readability required for automating operations.

An experimental prototype expert system called YES/MVS has been developed and installed at IBM's Thomas J. Watson Research Center. YES/MVS encodes knowledge of operations in several domains in logically distinct expert systems, including JESQ, an expert system that actively manages operating system queue space continuously in realtime. The design and testing of continuous realtime expert systems encompasses requirements not normally addressed in session-oriented consultative expert systems, particularly with regard to program control and the maintenance of an internal model

of the environment that is consistent with the actual state of the world. YES/MVS is among the first expert systems that operates continuously in realtime and exerts active control over its environment, and is also among the only such systems that have been operated and tested under real conditions. JESQ runs regularly at the IBM Yorktown Computing Center and has been informally approved by operations staff. The system runs in advisory mode (gives advice for operator review) during prime shift, and fully authorized (actually submitting commands to the target system) at night.

JESQ models the surface behavior of the operator, encoding operational policies in discrete knowledge base rules where possible. JESQ represents a fusion of the heuristic knowledge of several operators at the Yorktown Computing Center, and, to a lesser extent, encodes the knowledge of operations managers and systems programmers as well. JESQ represents an attempt to establish a central repository for queue space management policy that gets executed directly rather than by human operators. The hypothesis is that automating computer operations with expert systems techniques can potentially increase the availability and performance of existing computing resources, promote the productivity of operations staff, reduce operational labor requirements, and provide a partial solution to the problem of high turnover in operations personnel while providing a management tool for enforcing and reviewing policy that offers better management control through standardization.

More generally, this thesis provides the following contributions:

- An alternative methodology for managing large computing installations was suggested.
- General issues that arise in the construction of active, realtime expert systems (as contrasted with more traditional session-oriented expert systems) were identified, and strategies and mechanisms for dealing with some of these in the queue space management domain were described. Many of the methods employed in JESQ to accomplish realtime, active control may be applicable in other process control expert systems domains. An abstract problem description was provided to facilitate the identification of domains that are similar to the domain of queue space management.

- One fully-implemented architecture which integrates decision-making and acting in a realtime, multi-agent environment that is sufficiently complex to discourage explicit modelling of all interactions was described.

- An account of the methodology under which JESQ was developed was provided, supplementing the relatively sparse case data on the construction of expert systems in industrial environments. Of particular interest, the difficulties of testing an expert system in a complex, realtime environment were illuminated.

This work is intended to serve as an existence proof, establishing the feasibility of employing expert systems techniques in process control environments. The current implementation of JESQ successfully resolves JES queue space problems under test conditions in the real environment, and should provide the simplicity of representation required for continued review and maintenance of the knowledge base. However, it does not implement a comprehensive "theory of queue space management" that need only be parameterized. Rather, JESQ is collection of declarative programming constructs that conforms more closely to the format of an installation run book than its procedural-code equivalent.

Some elements of an alternative approach that approximates a "theory of JES queue space management" were presented, under which the event-response rules in JESQ were embedded in a more global framework that better captures the semantics of space management action scheduling than the general shell in which JESQ is implemented. This served as a vehicle for illuminating some of JESQ's shortcomings (although none of these represent serious flaws in terms of its required performance) and raised questions about both short and long term directions in realtime process control expert systems research.

While JESQ and other expert systems within YES/MVS have established the feasibility of automating operations using expert systems techniques, more research is required to produce a framework that provides for the encoding of process control specifications that are free of program control constructs. Perhaps the construction of several such applications will illuminate their important com-

monalities, leading to the development of a general tool. However, the continued application and adaptation of a preconceived general tool across several related domains also represents a feasible approach.

7. REFERENCES

[1] De Jong, K., "Intelligent Control: Integrating AI and Control Theory", *IEEE 1983 Conference on Trends in Applications*, 1983.

[2] Hong, S.J., "Knowledge Engineering in Industry", *Proceedings of Japan Systems Science Symposium*, 1984.

[3] Shortliffe, E.H., *Computer-Based Medical Consultations: MYCIN*, Elsevier, N.Y., 1976.

[4] Duda, R.O., Gaschnig, J., Hart, P.E., Konolige, K., Reboh, R., Barrett, P., and Slocum, J., "Development of the PROSPECTOR Consultation System for Mineral Exploration", Final report, SRI projects 5821 and 6415, SRI International, Inc., 1978.

[5] Buchanan, B.G., and Feigenbaum, E.A., "DENDRAL and Meta-DENDRAL: Their Applications Dimension", *Journal of Artificial Intelligence 11*, 1978.

[6] Duda, R.O., and Reboh, R., "AI and Decision Making: The Prospector Experience", *Proceedings of the NYU Symposium: Artificial Intelligence Applications for Business*, 1983.

[7] Barr, A. and Feigenbaum, E.A., eds., *The Handbook of Artificial Intelligence*, William Kaufman, Inc., 1982.

[8] Fox, M.S., Lowenfeld, S., and Kleinosky, P., "Techniques for Sensor-based Diagnosis", *Proceedings, IJCAI-83*, 1983.

[9] Wright, J.M., and Fox, M.S., "SRL/1.5 User Manual", Robotics Institute, Carnegie-Mellon University, 1982.

[10] Wesson, R.B., "Planning in the World of the Air Traffic Controller", *Proceedings of IJCAI-5*, 1977.

[11] Nelson, W.R., "REACTOR: An Expert System for Diagnosis and Treatment of Nuclear Reactor Accidents", *Proceedings of AAAI-82*, 1982.

[12] Chester, D., Lamb, D., Dhurjati, P., "Rule Based Computer Alarm Analysis in Chemical Process Plants", *IEEE Micro-Delcon '84: Proceedings, the Delaware Bay Computer Conference*, 1984.

[13] Fagan, L.M., "VM: Representing Time-Dependent Relations in A Medical Setting", PhD Thesis, Stanford University, 1980.

[14] Wright, P.K., Bourne, D.A., Colyer, J.P., Schatz, G.S., Isasi, J.A.E., "A Flexible Manufacturing Cell for Swaging", *Mechanical Engineering*, 1982.

[15] Forgy, C.L., "OPS5 User's Manual", CMU-CS-81-135, Dept. of Computer Science, Carnegie-Mellon University, 1981.

[16] Versonder, G.T., Stolfo, S.J., Zielinski, J.E., Miller, F.D., and Copp D.H., "ACE: An Expert System for Telephone Cable Maintenance", *Proceedings of IJCAI-83*, 1983.

[17] McDermott, J., "R1: A Rule Based Configurer of Computer Systems", *Artificial Intelligence* 19, 1982.

[18] McDermott, J., "XSEL: A Computer Sales Person's Assistant", *Machine Intelligence 10*, J. E. Hayes, D. Michie, and Y-H Pao, eds., J. Wiley and Sons, New York, 1982.

[19] Griesmer, J.H., Hong, S.J., Karnaugh, M., Kastner, J.K., Schor, M.I., Ennis, R.L., Klein, D.A., Milliken, K.R., Van Woerkom, H.M., "YES/MVS: A Continuous Real Time Expert System", *Proceedings of AAAI-84*, 1984.

[20] Pasik, A. and Schor, M., "Table Driven Rules in Expert Systems", *SIGART Newsletter*, No. 87, 1984.

[21] Guido, A.A, "Unattended Automated DP Center Operation: Is It Achievable?", *European GUIDE Proceedings*, 1983.

[22] Hayes-Roth, F., Waterman, D.A., Lenat, D.B., eds., *Building Expert Systems*, Addison-Wesley, 1983.

[23] McDermott, D., "Planning and Acting", *Cognitive Science 2*, 1978.