AN EXPERT SYSTEM ARCHITECTURE FOR FAULT-TOLERANT CONTROL IN UNSTABLE DOMAINS

BY

BRENT CHRISTOPHER SPILLNER

B.S., University of Illinois at Urbana-Champaign, 1998

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2000

Urbana, Illinois

Abstract

Modern expert systems are capable of rivaling or exceeding the performance of human experts at a number of challenging decision-making tasks. Their imperviousness to fatigue and potentially superhuman reflexes make them exceptionally suitable for the realtime supervision and control of complex systems. However, they cannot be deployed in place of human supervisors without strict guarantees of high reliabilty and near-constant availability. Although conventional techniques for the design of fault-tolerant computer systems can address these concerns, they miss opportunities to integrate information about perceived or potential failures into the expert system's standard reasoning process or to take advantage of that parallelism and extensibility inherent in convential expert systems to simplify failure avoidance and recovery protocols.

This thesis presents an innovative architecture for adding fault-tolerance capabilities to an existing expert system, with minimal disruption of the existing reasoning process and knowledge base. The design is verified by application to the Minerva-DC expert system for the shipboard damage control domain, but the techniques discussed should be applicable to a wide range of supervisory control systems and reasoning techniques.

To my parents.

Acknowledgements

This thesis would not have been possible without my advisor, David C. Wilkins, who guided my search for a thesis topic and supported this and my other research both intellectually and financially. Thanks to my CS 433 professor, Ravi Iyer, for initially stirring my interest in fault-tolerant systems. Thanks to current and former members of the Knowledge-Based Systems Group at the University of Illinois at Urbana-Champaign, for their comments, suggestions, and support of this research. In particular I would like to thank Vadim Bulitko, Adam Boyko, Satish Shankarappa, and David Fried for their insightful commentary during the Minerva-6 and Minerva-6/FT design processes.

This research has been partly sponsored by ONR Grant N00014-95-1-0749 and NRL Contract N00014-97-C-C2061.

Contents

Chapter

Appendix

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Fault-Tolerant Supervisory Control

Supervisory control refers to the task of observing some system or process through a set of sensors and attempting to influence its state or progress via the operation of selected members of a set of actuators. Supervisory control problems for which successful expert systems have been developed include industrial process control [GAMdP98, FYR98, Liu96] manufacturing [MZG+95, Kim95], power plant supervision [AFSV98], offshore oil production [KBE97], and aerospace systems [PGG+98]. In all of these domains, the systems being supervised and controlled may at times be too remote or hazardous for humans to access, making the development of autonomous automated supervisory control systems highly desirable. Even when the systems being supervised and controlled may be accessed directly or controlled from afar, automated supervisory control systems are normally more cost effective, and possibly more reliable, than human supervisors [Ste91].

In many important supervisory control domains, including the ones listed above, im-

proper or intermittent supervision and control can be extraordinarily costly or even life-threatening. In extremely dangerous and unstable domains such as shipboard damage control, there is a very significant risk that components of the supervisory control system itself will suffer unanticipated failures. These systems often need to be physically distributed and capable of decentralized operation in order to guard against the possibility that crises in one physical region of the control domain may disrupt the supervisory control system's ability to manage other regions of the domain. In these applications, a great deal of care must be put into the design and evaluation of fault tolerance mesaures for the supervisory control system [Ack87, PDK95].

1.2 The Shipboard Damage Control Domain

Shipboard damage control is the task of minimizing the adverse impact of physical damage to a naval vessel's ability to perform its current missions. It deals primarily with the containment of fires, flooding, smoke, and system failures, and the restoration of critical shipboard systems as quickly as possible with minimum loss of human life. It is usually the most personnel-intensive capability of a modern warship, and efficient management of damage control operations requires highly-skilled individuals with years of experience and intensive training. In the interest of improving personnel safety and reducing operating costs, the U.S. Navy is currently developing automated crisis detection and response systems to automate the most common damage control activities aboard its next generation of ships. The sheer complexity of these systems and the aggressive manning reductions for these ships requires the development of a reliable, effective automated supervisory con-

trol system for the shipboard damage control domain, to assist or even replace human supervisors. One effort in this area is the extension of an existing system for performance evaluation of human damage control supervision students to meet the more stringent requirements of an autonomous shipboard system, being conducted by the Knowledge-Based Systems Group at the University of Illinois at Urbana-Champaign [WS97].

1.3 Automated Approaches

The ability of computer systems to monitor their environments for long periods of time without becoming distracted or tired and to respond instantly to abnormal situations seems to make them well-suited to supervisory control applications. However, they must demonstrate very high levels of reliability before they can be trusted in the most important and dangerous domains [PDK95]. Many automated supervisory control systems rely on analytic models of "proper" system behavior derived from inviolable physical laws. When the system's observed behavior disagrees with the predicted behavior by a significant amount, separate analytic methods are used to attempt to diagnose the fault and possibly find a corrective action. This approach is effective for relatively simple, well-understood systems with easily measured properties, but breaks down when there is significant uncertainty in the sensory inputs and may not be possible in complex domains that are not easily modeled. Most analytic failure diagnosis and correction algorithms can be misled by multiple simultaneous failures, severely limiting the spectrum of fault situations such systems can tolerate [Ack84, Ise84].

Approaches involving connectionist networks and machine learning have been success-

ful at overcoming the limitations of analytic methods, but in general they are unable to provide explanations for their behavior or take advantage of the comprehensive policies and operating procedures that have been developed by human experts for the most complex and important supervisory control domains, or even to prove that all actions they take are in accordance with such doctrine. The training time for such systems is usually prohibitive, and adequate training examples may not be available for very rare but serious types of failures. Rule-based expert systems overcome these limitations, at the cost of requiring a complex set of rules to be specified by experts in the domain. For domains for which exhaustive doctrine already exists, however, such as nuclear reactor management or shipboard damage control, this is not a significant hindrance [Fra90].

1.4 Suitability of Expert Systems

Fortunately, the powerful inference and knowledge representation schemes of modern expert systems make them very well suited to the development of fault-tolerant systems. Information about potential or experienced faults can be directly integrated into the application's knowledge base, and reasoning about failure conditions can be performed simultaneously with the other reasoning tasks. Additionally, expert systems with the ability to monitor and critique the actions of other agents in the same domain can use that critiquing mechanism to evaluate their own performance or that of their peers in a redundant system, and thus potentially detect errant behavior due to software defects. Coupled with reliable mechanisms for storing knowledge and interacting with the environment, as well as a rule base that incorporates the possibility of component failures and erroneous information,

such expert systems should be able to handle a wide variety of hardware and software failures in a graceful and predictable manner.

Expert systems for fault diagnosis and recovery have been successfully deployed in a number of demanding supervisory control environments. However, these systems have generally dealt only with faults in the systems they monitor, rather than their own components and have not been physically distributed. In contrast, an expert system for managing shipboard damage control on a combat vessel must be able to operate successfully in a highly unstable environment in which many pieces of equipment, including components of the supervisory control system itself, are likely to be physically damaged or to lose electrical power. The size of a naval vessel and the likelihood that disasters may sever communication channels between different parts of the ship require that an effective supervisory control system be completely replicated in physically separated areas, with each replica able to manage local crises by itself in the event of communication interruptions.

This thesis presents an architecture for the construction of distributed expert systems able to diagnose and work around a wide variety of faults in themselves or the external environments they monitor and control. It also describes the way in which this architecture was employed in the adaptation of the existing Minerva-5 expert system to meet the fault-tolerance requirements of the shipboard damage control domain. Finally, it analyzes the performance of the extended system on simulated faults and the cost of applying these techniques to other expert systems.

1.5 Minerva

Minerva is a family of expert system shells developed by the Knowledge-Based Systems Group at the University of Illinois at Urbana-Champaign [Bul98]. All versions of Minerva share a blackboard architecture, a deliberate-schedule-execute reasoning cycle that allows explicit separation of domain, scheduling, and straegy knowledge, and an emphasis on the ability to provide advice, critiques, or performance evaluations for external agents (e.g. humans or other expert systems) attempting to solve the same problem. Early versions were designed for non-realtime applications such as offline medical diagnosis [PTW91, PTDW92]. Minerva-4 was designed from the ground up for realtime operation, with supervisory control applications specifically in mind [Bul97]. Minerva-5 extended Minerva-4 with more powerful environment prediction and state evaluation mechanisms and more flexible operating modes, and was the first version of Minerva to be deployed in a "production" supervisory control system, as part of the DC-TRAIN immersive damage control training simulator [Bul98]. Minerva-5 has been demonstrated to rival or excel domain experts at managing shipboard damage control scenarios in a simulated environment. However, it contains only rudimentary failure diagnosis and recovery rules and does not support distributed operation, making it unsuitable for unsupervised operation and threatening its utility in a real-world environment.

The current effort to develop Minerva-6 and Minerva-6/FT addresses these shortcomings. Minerva-6 duplicates the functionality of Minerva-5, but features a modular internal architecture that allows selective parts of the reasoning process to be upgraded or augmented without disturbing functionally independent components of the system. Minerva-

6/FT is an extension to Minerva-6 which takes advantage of this modular architecture to provide distributed, fault-tolerant reasoning using the principles and techniques described in this thesis. Minerva-DC is the name given to a version of Minerva-6/FT specialized for use in the shipboard damage control domain, being developed as a prototype supervisory control system for automated crisis detection and suppression systems aboard low-manpower warships of the future.

In early versions of Minerva, the flow of information and control between the various reasoning phases followed a rigid path, and the boundaries between these phases were occasionally blurred, making the addition of originally unanticipated features a cumbersome task. For Minerva-6, a new abstract blackboard interface was created and a new modular paradigm was established. Each of the reasoning phases is carried out by one or more independent modules, which are not allowed to make any assumptions about the existence of other modules or their own autonomy over any phase of the reasoning process, and which can only communicate with other modules by proposing changes to the shared blackboard. Special blackboard maintenance modules track these proposals and ensure that only mutually consistent changes are applied to the blackboard. This architecture makes it easy to incorporate distributed operation, modular redundancy, or even dynamic reconfiguration into the system without modifying any individual modules, which has proven invaluable in the implementation of Minerva-6/FT. Figure 1.1 illustrates the different "roles" filled by Minerva-6 modules and their interrelationships. A detailed description of each module role can be found in Section 4.1 of this thesis.

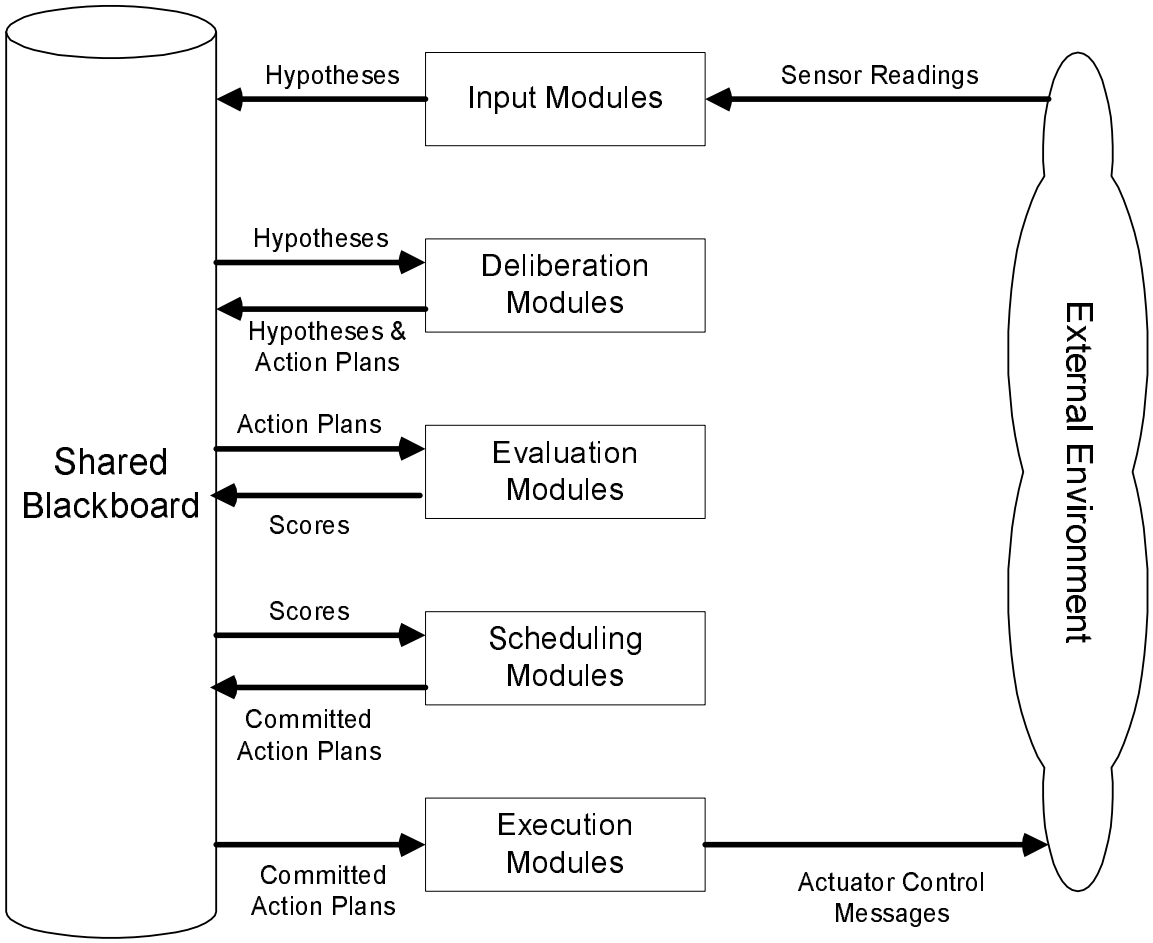

Figure 1.1: Minerva-6 data flow diagram

Chapter 2

Related Efforts

The literature on automated control systems is rife with examples of expert systems used to diagnose or cope with failures in complex systems, but relatively few of these systems are designed to survive failures of their own hardware or software components. Two novel approaches to the problem are the Fermentation Process Control System (FPCS) developed at the Tangshan Institute of Technology [Liu96] and NASA's New Millenium Remote Architecture (NMRA) project [PGG+98]. Both of these efforts revolve around a modular reasoning architecture into which rule-based systems for fault detection and recovery may be introduced without disruption of the core supervisory control algorithms. This is the same foundation that the Minerva-6/FT architecture employs, although Minerva-6/FT goes further by addressing the need for physical replication and decentralized operation of the reasoning system.

The Fermentation Process Control System comprises two separate knowledge bases, one for process control and another for fault diagnosis and recovery techniques. A strategy layer within the fault base continually evaluates the system state and ascribes appropriate

relative priorities to overall system performance and system stability, and a specialized inference engine uses these priorities to guide the reasoning process. All internal and external system parameters are continuously logged to a reliable external database, to allow for rapid recovery of the system state after a hardware or software failure. When the fault base rules detect abnormal or inconsistent sensor readings, they analyze a rule-based model of the controlled process' structure and failure modes and hypothesize about possible component failures. The strategy layer assigns a level of importance to each possible failure, and adjusts the system reasoning strategy accordingly. When a permanent failure is detected, fault recovery rules are used to modify the control equations in the process control rule base to reconfigure system behavior. By including a model of the FPCS itself within the rule base, the system is even able to detect erroneous internal state or rules and purge them from the system [Liu96].

When deployed, the New Millenium Remote Agent architecture will be the first automated system to be given complete autonomy over a spacecraft. Its basic design principle is the union of a traditional procedural control system with a deductive expert system which continually monitors the observed system state for evidence of failures. When a component failure is detected, fault recovery rules are used to modify the procedural control system to bypass or restart the failed components. The system uses flexible resource allocation strategies to degrade mission performance as gracefully as it can [PGG+98].

Chapter 3

Approach

3.1 Failure Model

The failures the Minerva-6/FT architecture strives to overcome may be classified by the role of the failed component in the overall system and the behavior of the component after the failure. Components are classified as either *internal*, meaning they form part of the hardware platform which runs the expert system, *external*, meaning that they are part of a sensor providing input to the expert system or an actuator carrying out commands from the system, or *communication channels*, meaning they relay information between internal and external components. Communication channel failures can only be distingushed from external component failures when there is more than one communication channel between the supervisory control system or the sole communication channel to the component also acts as a communication channel for other components. The behavior of components after failure is classified as either *silent*, meaning that the failed component ceases to interact with the expert system in any way, or *Byzantine*, meaning that the failed component may

Component	Type of Failure	
	Silent	Byzantine
Internal	Modular redundancy	Sophisticated scheduler; peer monitoring with the critiquing mechanism
External	"Expected Response" system for action followup	"Expected Response" system for action followup
Communication Channel	Use of Oracle 8's Multimaster Replication to keep each network partition "alive"	Use of error-detecting network protocols

Table 3.1: Failure Avoidance Techniques in Minerva-6/FT

behave in an arbitrary fashion. Each of the six different types of failures in this taxonomy is handled by a different facet of the fault tolerance architecture. Table 3.1 summarizes the approach Minerva-6/FT uses to attempt to prevent each type of the component failure from causing a complete system failure.

3.2 Design Goals

Although resistance to failure is the primary goal of the proposed expert system architecture, that is not the only factor driving its design. In many cases, including that of Minerva-DC, it is desirable to be able to extend an existing system or prototype to deal with potential component failures without requiring that the core program logic or associated development tools and utilities be completely reimplemented. Accordingly, the first step in the development of Minerva-6/FT was the decomposition of the reasoning process into functionally orthogonal modules, with no module allowed to make assumptions about

the reliability or availability of any of its peers.

Another key goal of the Minerva-6/FT design is to make use of off-the-shelf components wherever possible. The development and testing of low-level fault detection and recovery protocols can be extraordinarily time consuming, and this work has already been done in a number of commercial products. A major design goal of Minerva-6/FT is the use of off-the-shelf components, such as fault-tolerant databases and networking systems, to provide reliable storage and transportation of data wherever possible.

3.3 Silent Failures of Internal Components

Silent component failures, even within the supervisory control system itself, are hardly improbable in chaotic situations such as a spacecraft after a collision with another body or a warship during a crisis, although admittedly crises aboard warships are far more common than collisions in space. Rapid recovery of these components may not be possible, especially if they have been physically damaged or if the local electrical distribution system has been disrupted. The only practical way to reinforce a supervisory control system against this sort of failure, therefore, is to provide redundant copies of each of its components, and to require each to send out a periodic "heartbeat" message to its peers so that silent failures of these components or the communication channels may be rapidly detected.

There are two ways to make use of redundant components in the system: one primary instance of each can handle all of the work, with a backup instances on "standby" to take over the task if a problem with the primary instance is detected; or the task can be divided up into approximately equal pieces which are assigned to individual instances

of the redundant component, with the load dynamically rebalanced whenever an instance failure is detected. The design of Minerva-6/FT follows the latter approach, in the interest of minimizing the amount of work disrupted by an instance failure and of making optimal use of available resources even before failures occur. The disadvantage of this approach is that it requires a high level of coordination between the cooperating instances of each component, but the modular decomposition and rule-based nature of the Minerva-6 core architecture make this coordination easy to achieve.

Minerva-6/FT's basic approach to this coordination problem is to divide the domain to be supervised and controlled into a number of (mostly) disjoint zones of jurisdiction, and to then assign one or more zones of jurisdiction to each instance of a redundant component. The decomposition can be along physical or functional dimensions, the zones need not be completely disjoint if the expert system can correctly monitor and account for the possibility of independent "agents" carrying out actions in the domain it is supervising, as Minerva-5 and Minerva-6 can, but a disjoint decomposition is still desirable in order to minimize the amount of computation and communication necessary to provide a given level of domain coverage. In the shipboard damage control domain, special "reinforced frames" dividing the vessel into watertight sections for the purpose of damage isolation provide a natural domain decomposition with very little chance of a single crisis crossing zone boundaries, so they were chosen as boundaries for the jurisdiction zones.

In order to ensure that all blackboard items are consistent between nodes in the distributed system and recoverable after a node failure, Minerva-6 uses an external database to hold a log of all proposed and approved changes to the blackboard. Coordination modules monitor the proposed changes and approve mutually consistent sets of changes, and

all reasoning modules in the supervisory control system monitor the database and apply the approved changes to their internal snapshots of the shared blackboard. By using an database management system with support for replication and consensus protocols, the blackboard will be preserved after single-site failures and consistent among all modules in the distributed reasoning system.

3.4 Byzantine Failures of Internal Components

Byzantine failures are much more difficult to detect than silent failures, and because the failed components may be making arbitrary changes to the environment and the blackboard before a failure is detected, they are generally more difficult to recover from as well. Fortunately, the problem is exactly equivalent to that of monitoring the actions of an external agent, evaluating its performance, and suggesting corrections, which Minerva-5 and many other expert systems are already equipped for in the form of a critiquing mechanism. Minerva-5's critiquing mechanism to monitor and provide feedback to human supervisors learning to supervise and control the domain by themselves, but it is also perfectly applicable to the problem of monitoring other Minerva-5 instances and looking for Byzantine failures.

In order to guard against the possibility of a failed instance of the critiquing subsystem announcing that is has detected Byzantine failures in properly working modules, scheduling rules have been added to Minerva-6 to prevent the `deprecate_instance` (i.e. ignore all output from the instance, in order to disable instances with Byzantine failures) action from being executed unless it has been proposed by at least two different Minerva-6 scheduler

instances. The scheduler itself employs a modular redundancy scheme to guard against scheduling failures causing proposed actions to be erroneously approved for execution. At least three instances of the scheduler are kept running at all times, preferably on different machines, and a majority of the active schedulers must agree to commit an action plan before the execution modules will carry it out.

Finally, because there may be an arbitrary number of Minerva-6 modules fulfilling any role except scheduling, a measure of insurance against software design or implementation errors may be acheived by using a variety of different modules, preferably even developed by different teams, for the various reasoning tasks. The propose-evaluate-schedule paradigm allows for seamless integretation of new reasoning modules and intelligent selection of the best course of action, which allows a measure of "algorithmic redundancy" to be incorporated into the system. A complete N-version algorithmic redundancy approach can be implemented by simply adding new implementations of various modules to the system, while a recovery block approach could be implemented by letting older versions of the modules reason in parallel with newer versions. A system employing algorithmic redundancy in this fashion has not yet been developed, but the Minerva-6/FT system architecture was chosen in part because it supports this as an area for future improvement.

3.5 Silent Failures of External Components

Silent failures of external components can only be detected by their failure to provide information on an expected schedule or their failure to respond to commands when an appropriate communication channel appears to be intact. In Minerva-6/FT, this task is

relegated to the input and execution modules, which propose hypotheses detailing the failure symptoms when expected reports or acknowledgements are not received from external components.

Although it is important to be able to remember the components which have been classified as failed and adjust action plans accordingly, it is equally important to be able to diagnose the cause of the failure from observed failure patterns and use that information to predict future failures or infer information about the state of the world. For example, simultaneous failure of all components powered by a particular electrical switchboard can be used as evidence for a crisis in the neighborhood of that switchboard. The external components of the domain, as well as their interdependencies, failure modes, and failure probabilities, are represented by a Markov model inside the supervisory control system. For Minerva-DC, the most basic aspects of the failure model are currently encoded in a small set of domain rules, and a much more powerful Extended Petri Net representation is being developed. This failure model is then integrated into the rest of the Minerva-6 system as both a deliberation module to propose hypotheses about the likelihood of specific failure and as an evaluation module to consider action plans in the context of their dependency on possibly failed system components.

3.6 Byzantine Failures of External Components

Byzantine failures of components providing input to the supervisory control system (such as sensors) are detected by attempting to corroborate all sensor findings with information from other nearby sensors. The Minerva-DC system contains a Crisis Recognizer deliberation

module that uses this technique, referred to as Neighborhood Crisis Detection, to determine appropriate confidence factors for sensory inputs.

Byzantine failures of components carrying out orders from the supervisory control system are detected by monitoring input components in the neighborhood of those output components for the expected results of each order. In Minerva-6/FT this is done at a low level inside the execution modules, which contain domain rules describing the expected effects of each potential action and a set of hypotheses to assert if confirmation of an expected effect is not received within a specified amount of time. For example, Minerva-DC contains a rule that when an automated fire suppression system is deployed within a compartment, the CO_2 concentration within that compartment should quickly stop increasing. Higher-level followups to entire action plans may be handled inside the deliberation modules which proposed the plan.

3.7 Silent Failures of Communication Channels

The Minerva-6/FT architecture requires that the underlying communication network satisfy both symmetric and transitive properties, i.e. if network node A is able to send messages to node B, then node B is also able to send messages to node A; and if node A is able to send messages to node B and node B is able to send messages to node C, then node A is also able to send messages to node C. Byzantine failures of a communication channel (discussed below) may cause it to temporarily appear to violate these properties, but as long as all messages that should be deliverable according to these properties can be delivered within a finite number of attempts, the distributed instances of the supervisory control

system can remain synchronized with each other. Minerva-6/FT is currently designed to communicate using TCP/IP over an Ethernet infrastructure, a network architecture that does in fact satisfy both of the required properties.

Together, the properties of symmetry and transitivity imply that every node X in the distributed system is in two-way communication with every other node in the largest spanning tree containing node X in an undirected graph of the network topology. Silent failures of communication channels serve to remove edges from this undirected graph, possibly partititioning a connected set of network nodes into two disjoint sets. When the network is partitioned in this fashion, the components within each still have a common view of the blackboard, but the blackboards may not be synchronized across the partition. The system components within each partition initially see the partition as a simultaneous failure of all components in the other partition, and adopt responsibility for the zones of jurisdiction that had been assigned to the unavailable components. Figure 3.1 shows the behavior of the modules on each machine after a network partition.

For a silent failure, the network is truly severed and all external components in the other partition will remain inaccessible as well. After taking over responsibility for the zones of jurisdiction handled by the inaccessible internal components, the components in each partition will attempt to handle the supervisory control task for the inaccessible network partition, which is the appropriate behavior for the case where the problem really is simply internal component failure. All attempts to communicate with the external components in the inaccessible partition will fail, however, and the queue of problems and tasks to be executed in that partition will quickly be drained, so that reasoning tasks for the inaccessible portion of the network will quickly stop consuming computational resources,

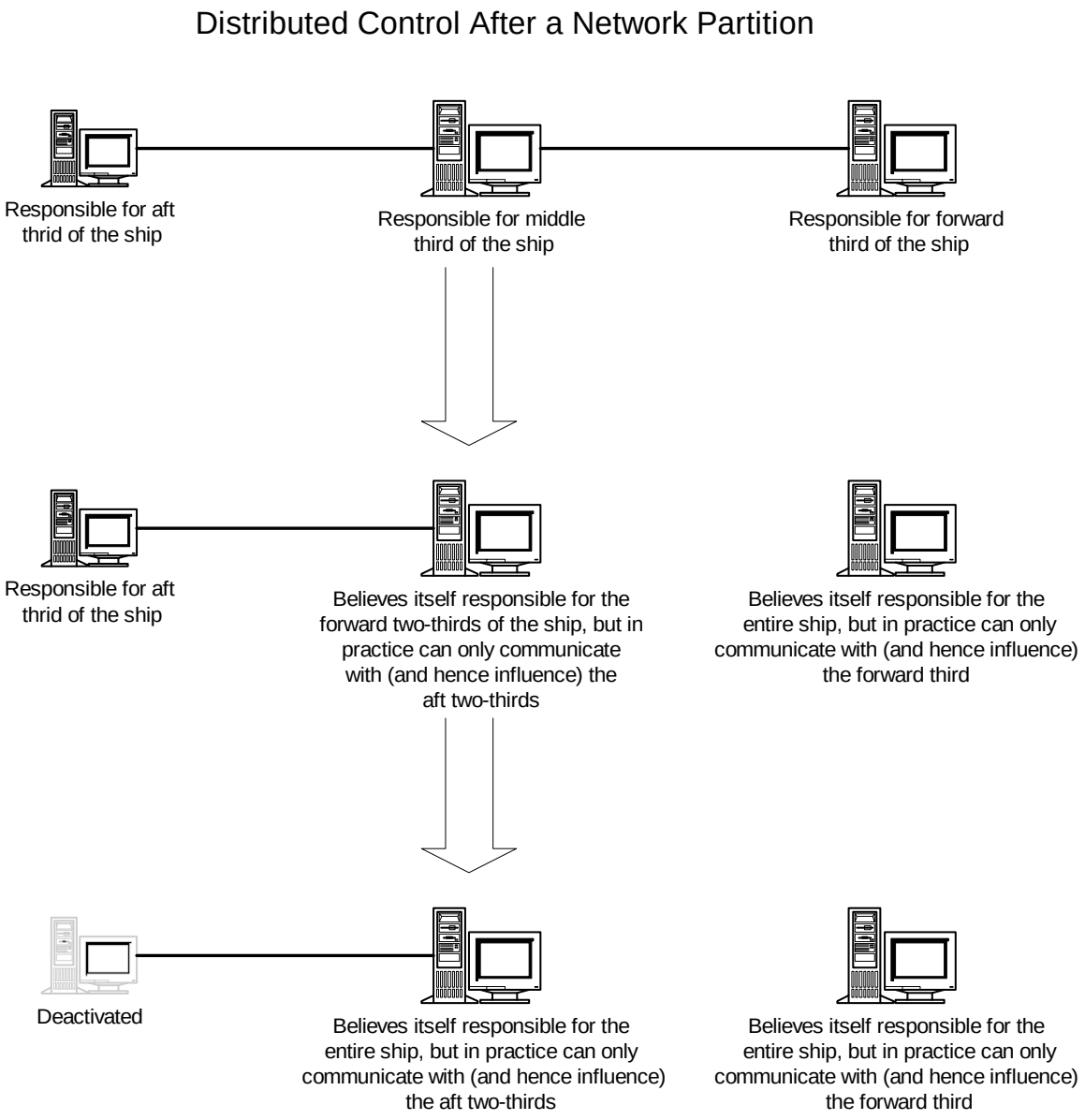

Figure 3.1: Minerva-6/FT Behavior After a Silent Network Failure

and the two partitions will efficiently confine their respective attentions to the accessible portions of the domain. A more detailed description of this process and result is given in Chapter 4 of this thesis.

3.8 Byzantine Failures of Communication Channels

One of two possible Byzantine failures of a communication channel is the corruption of data sent across it. This is an issue normally dealt with at very low levels, and the Minerva-6/FT design entrusts the communication protocols and database manager with the responsibility of detecting and retransmitting corrupted messages. Bad data which does escape these low-level checks is indistinguishable to the application from data generated by a Byzantine failure in an external component, and the mechanisms for dealing with such failures are automatically applied to data corrupted by the communication channel.

The other possible Byzantine failure of a communication channel is the periodic loss and restoration of the last remaining communication channel between two partitions of the distributed system. If the downtimes in the sequence are long enough, the shared blackboards in the two system partitions may become irreconciliably desynchronized. This is a very difficult problem with no general solution except the selection of one blackboard as "authoritative" and rollback of the other to a point before synchronization was lost. This is technically possible within the Minerva-6 blackboard architecture, but is undesirable for a number of reasons, including the inability of the rolled-back system to respond to crises until it "catches up" and the possibility that information will be lost. Instead, Minerva-6/FT opts to let the database replication engine unify the blackboards if it can, or leave the

system effectively partitioned if it can't. This latter option is distasteful as well, because it prevents the partitions from sharing information even though communication has been restored, but it is a far simpler and more conservative approach.

It is still important, however, to ensure that jurisdiction boundaries are properly maintained when the network is restored, so that there cannot be multiple instances of an internal component which simultaneously believe that they are responsible for a single zone of jurisdiction. Minerva-6/FT solves this problem by broadcasting all heartbeat messages and storing them in a database table that is not protected by a replication mechanism, so that heartbeat messages will still be shared when the blackboards have become desynchronized. Each internal component of the supervisory control system is then assigned a unique integer ID, and jurisdiction conflicts are resolved by assigning the contested zone of jurisdiction to the module having the lowest ID. This protocol is illustrated in Figure 3.2 and described in more detail in Chapter 4 of this thesis.

3.9 Design Verification

Design verification is perhaps the most important aspect of the development of fault-tolerant systems. Analytic verification is often intractable and almost always extremely expensive for even moderately complex distributed systems, although a few well-funded projects have experienced success [ORSvH95]. Accordingly, empirical design verification is an important part of the design process of most fault-tolerant systems, even when analytic techniques are also used [SS98]. Many expert systems already have simulator testbeds to faciliate features such as machine learning, which makes empirical design verification

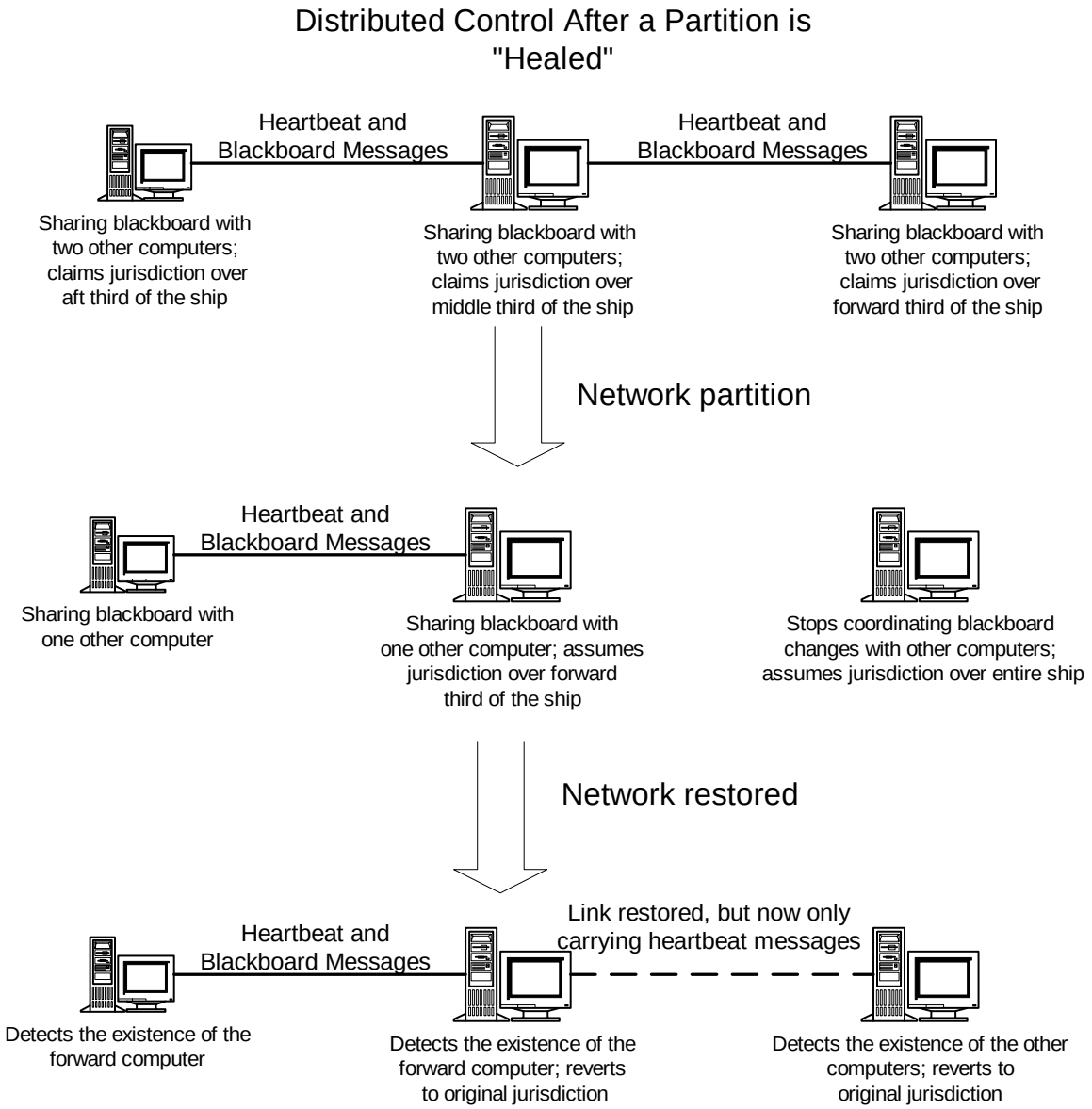

Figure 3.2: Minerva-6/FT Behavior After Byzantine Network Failure

even more attractive, but analytic techniques are also applicable to the expert system architecture described in this thesis.

Minerva-DC currently runs only in a simulated ship environment. Highly flexible scenario specification and simulation tools have already been developed and tested for Minerva-DC's instructional applications, and these were easily adapted (in most cases no changes were needed at all) to simulate arbitrary communication channel and external component failures. Silent failures of internal components are easily simulated by breaking power or network connections between hardware, while Byzantine failures of internal components are simulated by modifying the scheduling algorithm to select action plans for commission at random. In the next fiscal year a version of Minerva-DC will be tested on the ex-U.S.S. Shadwell, a decommissioned warship that has been specially modified to conduct trials of advanced sensors, actuators, and intelligent control systems for shipboard damage control. At the present time, however, the only available testbed is the artificial ship simulator.

The Minerva-6/FT system is objectively evaluated by the scoring facility of the existing Minerva-5 expert system, which has demonstrated the ability to accurately evaluate "novel" actions that its own rule base would not propose, allowing adaptive scoring and critiques of human experts or other expert systems. The shipboard damage control version of Minerva-5 already significantly exceeds the average performance of human experts on simulated crises using these prediction techniques, and thus can be trusted to accurately evaluate the performance of Minerva-6/FT [Bul98].

Chapter 4

Implementation Details

4.1 Minerva-6 Modular Architecture

The Minerva-5 architecture was demonstrably successful at several tasks, including shipboard damage control in a simulated environment, but it could not easily accomodate multiple state evaluation techniques or be decomposed into parallel components for use in a distributed system. The Minerva development team deemed these capabilities essential for fault-tolerant operation or adherence to realtime constraints, two significant research directions for the Minerva architecture. In order to support these capabilities and potential future extensions, the team decided to a completely new modular architecture for Minerva-6, with all module interaction arbitrated by a shared external blackboard.

The first step in the design of Minerva-6 was the decomposition of Minerva-5's reasoning process into five functionally orthogonal "roles", each of which could be assigned to one or more independent modules. Modules filling the "input" role are responsible for monitoring sources of information about the external environment and converting their

observations into findings or hypotheses on the blackboard. Modules filling the "deliberation" role are responsible for devising action plans which may improve the quality or viability of the external environment. Modules filling the "evaluation" role are responsible for predicting the impact of the proposed action plans on the external environment and its state evaluation metrics. Modules filling the "scheduling" role are responsible for using these predictions to rank the proposed action plans and occasionally select plans for execution. Finally, modules filling the "execution" role are responsible for generating suitable control messages or system outputs to carry out each of the selected actions.

The flow of information between these roles and the external blackboard is shown in Figure 1.1. The blackboard itself must be consistently viewed by all modules, which requires the introduction of two special roles, "hypothesis coordination" and "action coordination", which are responsible for ensuring that only a consistent subset of the changes proposed by other modules are actually applied to the blackboard. All consultation of and proposal of changes to the blackboard is done through an abstract interface library, allowing the external representation of the blackboard to be changed at any time. The current blackboard implementation is a log of all proposed and committed changes, stored in a relational database accessed through the ODBC (Open Database Connectivity) mechanism. The change log representation allows modules to come online or restart at any time and still obtain a complete history of reasoning activity, which is critical to recovery from internal component failures. The use of ODBC allows the underlying database engine to be changed without modifications to the Minerva source code, as well as seamless access to databases located on a different host than the accessing module. The structure of the external database is summarized in Table 4.1.

Table Name	Description
CoordinatorElection	Record of "votes" in the election of new coordination modules for the rest of the blackboard
MinervaBlackboardChanges	Log-style record of approved changes to the internal list of hypotheses. Records are appended to but never removed from this table, so one can view the state of the blackboard at any point in time.
MinervaCommittedActions	Log-style record of action plans selected for execution by the scheduling module.
MinervaDebatableActions	Log-style record of action plans open for consideration by evaluation modules.
MinervaDeprecatedActions	Log-style record of action plans that evaluators should no longer consider.
MinervaProgress	Log-style record of the reasoning progress of each participating module, used to synchronize the reasoning pipeline in a distributed system.
MinervaProposedActions	Log-style record of action plans proposed by deliberation modules.
MinervaProposedBlackboardChanges	Log-style record of hypothesis changes proposed by deliberation modules.
MinervaProposedScores	Log-style record of action plan scores proposed by evaluation modules.

Table 4.1: Representation of the Blackboard in the External Database

4.2 Jurisdiction Rules

Minerva-FT uses the concept of "zones of jurisdiction" to distribute work and responsibility throughout the system. The problem domain is statically divided into one or more zones of jurisdiction, with the zones chosen to be as nearly disjoint as possible. During system execution each module instance is responsible for 0 or more zones of jurisdiction at all times, subject to two constraints: that no two instances of the same module simultaneously have jurisdiction over any zone; and that there is some constant amount of time such that if no instances of a particular module fail during an interval of that length, then each zone of jurisdiction will be held by exactly one instance of the module at the end of the interval. An example of jurisdiction reassignment after communication channel and internal component failures is illustrated in Figure 3.1.

Module availability is monitored with a protocol first proposed by Chandra and Toueg [CT96] for detection of silent process or link failures in an asynchronous network. When a failure is detected, the remaining instances of the failed module can seek to claim each of its zones of jurisdiction by proposing hypotheses to that effect on the shared blackboard, with the jurisdiction awarded to the module whose hypothesis appears first. The blackboard provides a serialization between hypotheses, so that all modules can agree on who was awarded which zone of jurisdiction. By delaying jurisdiction claims by an interval proportional to the local system load, approximate load balancing can be achieved by this scheme. The exception to this protocol is when the hypothesis coordination module itself fails. When modules notice that no new hypotheses (including the "heartbeat" messages in the failure detection protocol) have appeared on the blackboard in the last few seconds,

they participate in a variant of Peterson's algorithm [Pet82] through the CoordinatorElection table on the blackboard (which requires no central arbitrating authority) to elect a new hypothesis coordinator.

If the apparent module failures are in fact due to a network partition isolating groups of modules from each other, the recovery protocols will be carried out in each isolated subnetwork. If the network is restored after the protocols have run to completion and jurisdictions have been reassigned, there is a potential for confusion or incorrect behavior. Unfortunately, little theoretical work has been done on protocols which operate properly after temporary network partitions, and the proposed jurisdiction system does not deal elegantly with this case. A future research goal is to develop a system capable of resolving conflicts and efficiently revising jurisdiction assignments after a partitioned network has been restored or new participants have been brought online for the first time. In the current system, however, network partitions must be "enforced" even after physical failures have been repaired, either through features of the networking hardware or by using an abstract blackboard interface that automatically ignores entries added by modules believed to be failed or inaccessible. The Minerva-6/FT system uses the latter approach.

An evaluation module which downgrades actions affecting zones outside the jurisdiction of the module instance proposing the action is used to enforce the jurisdiction concept for deliberation modules which do not handle jurisdiction issues themselves. Very urgent actions should still have very positive scores after the downgrade, so rapid responses to problems outside a module's jurisdiction can still be approved. This allows module instances to step in and manage critical events (e.g. a fire in a weapons magazine in the damage control domain) occurring outside their zones of jurisdiction without the need

to first verify the availability or failure of the presiding instance. This policy introduces the possibility that multiple commands will be issued to actuator systems to deal with a single problem, which could be a source of confusion or conflict. It is therefore assumed that the actuator systems will be able to deal intelligently with duplicate messages, and that the utility threshold at which an action plan can overload a jurisdiction rule will be set sufficiently high that only extremely beneficial and low-risk actions can be issued in this manner. The precise level at which the threshold should be set is of course domain-specific and a good candidate for machine learning or other refinement methods. The rules for ranking actions based on the concept of "jurisdiction" form part of the rule-based evaluator in Appendix C.

4.3 Resource Contention

The efficient allocation of available resources is a crucial problem for most supervisory control systems. Different subgoals within the overall goal of "effective supervision" may have conflicting immediate requirements. For example, one deliberation module might wish to shut down a malfunctioning actuator system, while another might need to use it to deal with some urgent problem. This is an issue that every supervisory control system must deal with at the domain knowledge level, but the modular nature of Minerva-6/FT makes the problem more acute, because each reasoning module has little awareness of the operation of the others. This problem is partially solved by the use of predictive evaluators (implemented in Minerva-6 as extended Petri networks) which track all committed action plans and consider proposed plans in the context of their effect on ongoing efforts. After

a network partition, however, these predictors may have imperfect knowledge about the environment, and it is possible that efforts in different partitions of the domain could interfere with each other.

For example, in the shipboard damage control domain the supervisory control system in one region of the ship may wish to halt the flow of water through the main firefighting supply system in order to repair a rupture, while the supervisory control system in another region of the ship may be relying on that water to fight a fire. Minerva-DC solves this problem by borrowing a technique from the domain doctrine itself to implement a "tag-out" procedure for important equipment. Tag-outs are implemented as normal actions, with the semantics of "reserving" a particular piece of equipment for a specified amount of time. If the tag-out is not renewed within that interval, it expires, allowing other deliberators to consider shutting down the affected actuator. Tag-outs in effect before the network is partitioned are known to modules in both of the isolated subnetworks, and actuator systems with internal support for the tag-out protocol can even enforce tagouts issued after two supervisory control systems able to control the actuator system lose their ability to communicate with each other.

4.4 Critiquing Subsystem

Minerva-6 provides a critiquing facility through the use of special input and execution modules. The input module monitors the appropriate external for actions to be critiqued and proposes equivalent action plans to the blackboard, just as a standard deliberation module might, but with a special flag marking them as actions to be critiqued, rather than

executed. The execution module tracks the computed utilities of these specially marked action plans and sends significant changes back to some external entity, which presumably formats appropriate critiques and offers them to a user or supervisor.

Although this facility is useful simply to improve usability and acceptance of a supervisory control system [She98], it is also a powerful tool for detecting and dealing with Byzantine failures in the reasoning system itself. Each instance of the reasoning system can be continuously critiqued by its peers or even older versions of the expert system, allowing erratic or harmful behavior can be rapidly detected and trends to be recognized. The critiquing modules can then hypothesize that the erratic modules are suffering Byzantine failures with a confidence proportional to the severity of the critique. Once the confidence of this hypothesis has been pushed above a very high threshold by multiple critiquers (to guard against the effects of Byzantine failure in the critiquers themselves), the module can be designated as "failed" and its jurisdictions reassigned just as though it had been detected by the silent failure detection protocol described in Section 4.2 of this chapter.

Minerva-DC uses the critiquing and scoring mechanism from Minerva-5 to evaluate its own performance. Minerva's critiquing mechanism uses the internal problem-solving rulebase to formulate the critique, so it is unlikely to rate an instance of the same version of Minerva very poorly. Minerva-5's damage control rulebase has been thoroughly evaluated in fault-free scenarios, while Minerva-DC's is still under development, making Minerva-5 a more reliable critiquing agent. Minerva-5's critiquing mechanism passes "novel" actions (i.e. ones Minerva-5 itself has not considered) to the predictive evaluators, so Minerva-5 should still generate reasonable critiques for actions which are more correct (e.g. ones which take detected or potential faults into account) than its own, allowing it to perform

effective critiques of Minerva-DC.

4.5 Oracle 8 and Blackboard Replication

Minerva-6 uses an abstract interface to communicate with a shared external blackboard, which is assumed to archive all blackboard operations in reliable storage. The abstract interface uses ODBC to access a relational database representation of the blackboard, which allows the physical database to be located on any machine accessible over a network and to use any of a number of third-party relational database engines. For fault-tolerant distributed operation, the external database must present a consistent view of its contents to modules on multiple machines, with serialized transactions and the ability to survive any combination of host or link failures. The survivability requirement implies that the blackboard must be replicated or at least reconstructable on all participating hosts, with all changes simultaneously propagated to all replicas and globally serialized. The challenges of implementing such a communication system are numerous, but fortunately these requirements are shared with a large number of distributed applications [BJ87, SS97, Sin97]. A number of well-tested third party database engines and communication libraries exist which meet these requirements; Oracle 8 Enterprise Edition was selected for Minerva-6/FT because of its automated replication features and efficient ODBC interface.

Each host in the Minerva-6/FT cluster contains a copy of an Oracle 8 database, distributed using what Oracle terms a "multimaster scheme with synchronous updates", meaning that each replica may be updated and all committed updates are simultaneously comitted at all locations. Oracle 8 uses a customized network interface that optimizes the

communication required for the underlying network architecture; to date Minerva-6/FT has only been implemented in a TCP/IP environment supporting broadcast between all hosts, which imposes a very high communication cost on the database transactions. Oracle allows applications to specify conflict resolution rules for conflicts that Oracle's default behavior cannot resolve (e.g. resynchronization after a network partition), but the Minerva-6 blackboard interface library is not sophisticated enough to support this sort of revision. Future refinements of Minerva-6 could remove this restriction, since the blackboard is cleanly abstracted from the reasoning code, but right now intricate conflicts cannot be resolved, forcing the shared database to split between the network partitions. This behavior is discussed in more detail in Section 4.2 of this chapter.

4.6 Action Followup/Failure Detection

One of the key innovations of the Minerva-FT architecture is the concept of "following up" on all committed actions. Rules in deliberation and execution modules can specify expected results for each domain action executed, a time by which the result should be detected, and a goal to evaluate if the result is not observed within the specified interval. For example, if an acknowledgement to an order is not received within a few seconds after the order is issued, the possibility of a communication channel or external component failure can be explored, and hypotheses about each potential failure can be posted to the blackboard to guide further reasoning. By moving this logic into the execution modules, deliberation modules can remain oblivious to the possibility that the actions they recommend may not be successfully carried out, although deliberation modules are welcome to specify expected

responses directly to provide higher-level fault detection and response.

The Minerva-DC execution module specifies both general constraints that different types of actions may imply (for example, all orders should be acknowledged, all requests for information should be answered) and specific constraints for individual actions (for example, the water level in a compartment should rise after it is order to flood it is acknowledged). Every time the execution module is "awakened" by the flow control system, it looks through the queue of expected responses for constraints that have not been satisfied within the given time limit. The exception handler goals for these unsatisfied constraints are then evaluated to propose alternative actions and/or log the anomaly to the blackboard for consideration by the domain-specific failure diagnosis modules. The rules comprising the Minerva-DC execution module are shown in Appendix A. Figure 4.1 is a flowchart which illustrates the high-level operation of this module.

4.7 Failure Diagnosis

Failure diagnosis is performed by special deliberation and evaluation modules which track the constraint failures logged to the blackboard. These modules contain a model of key internal and external components, their failure modes, and their interdependencies, generally encoded as a domain rule base or a Petri net. Information about anomalous observations or actuator behavior is fed in to the models as evidence of possible failures, and hypotheses about component failures that might explain this behavior are proposed to the blackboard with appropriate confidence factors. Other deliberation and scheduling modules can then use these hypotheses to guide their assessment of the environment and proposed action

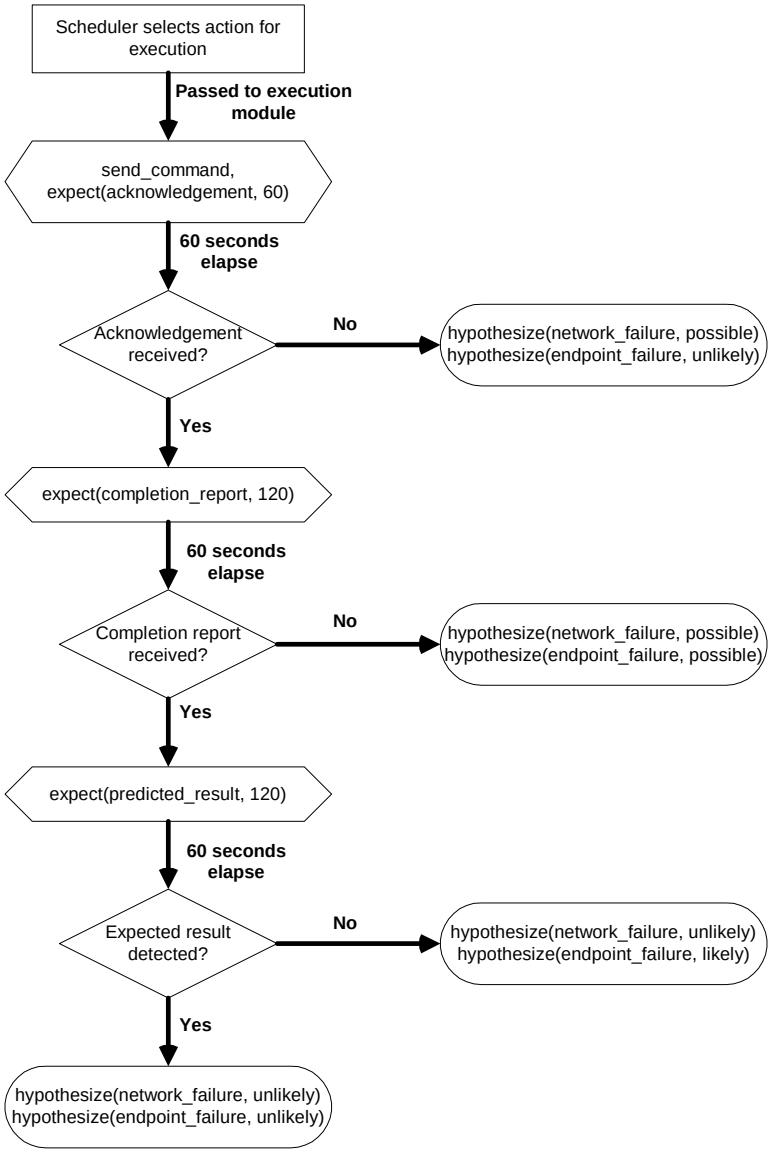

Figure 4.1: The expected response/countermeasure system.

plans. A simple rule-based evaluation module which uses information about possible system failures to rank action plans dependent on failed components is shown in Appendix C.

Minerva-DC contains a rule-based model for the shipboard electrical distribution system which is used to make decisions about the routing of casualty power connections, as well as a fuzzy logic-based model for sensor behavior, which is used to convert sensor readings into hypotheses about compartment states. These models double as failure diagnosis knowledge bases because they detail expected component behavior and relationships. In addition, the Petri networks used for predictive evaluation contain places corresponding to diagnosed system failures, allowing actions which would be hindered by these failures to be properly predicted as ineffective and rated accordingly.

Chapter 5

Performance Evaluation

5.1 Fault Simulation

In Minerva/FT, external component faults are most easily simulated by special rules in the input and execution modules of the reasoning system. Transient sensor and network faults are simulated by randomly ignoring or corrupting incoming information before it is posted as a finding on the blackboard, while transient actuator faults are modeled by randomly discarding selected action plans marked for execution and incoming acknowledgements from the actuators. Systematic failures of external components are modeled by asserting rules to ignore all incoming messages from and discard all outgoing messages to those particular objects. Loss of network connectivity or internal system components is modeled by simply unplugging the appropriate network or power connections. Transient software faults are modeled only by inducing random behavior in the system scheduler, although more complex failures could be introduced.

5.2 Experimental Design and Results

To evaluate the performance of the Minerva/FT architecture in the shipboard damage control domain, Minerva-5, Minerva-6, and Minerva/FT were each run on each combination of one of three damage scenarios and one of four fault patterns. The results are evaluated by comparing the number of crises remaining after 25 minutes, the success or failure of each system at avoiding "kill points" (catastrophic events that the ship cannot recover from, such as an explosion in a missile magazine), and the overall performance scores computed by Minerva-5's scoring subsystem. Each test was conducted using three Minervas simultaneously, and for the Minerva/FT tests the ship was divided into jurisdiction jones exactly matching the jurisdiction zones of the three repair lockers aboard an Arleigh Burke-class destroyer.

The three scenarios are "Combo" (medium-size fires fore and aft, with flooding aft below the waterline), "Blaze" (large fire in the ship's superstructure), and "Chaos" (multiple fire and flooding outbreaks almost simultaneously throughout the ship). The four fault patterns are Power (loss of one of three computers running Minerva and four of the ship's six firepumps), Network (no information crosses a reinforced transverse bulkhead near the center of the ship), Damage (loss of equipment and random noise from sensors and alarms in the aft part of the ship), and "None", no faults are explicitly introduced (although the simulator still models faults caused directly by the crisis). No correlation is made between the damage scenarios and fault patterns, but each is representative of the dangers threatening surface warships.

The results from these trials are summarized in Table 5.1. The fault patterns used are

Fault Pattern	Scenario	Unresolved Crises			Kill Point?			Overall Score		
		M-5	M-6	M-6/FT	M-5	M-6	M-6/FT	M-5	M-6	M-6/FT
None	Combo	6	1	1	No	No	No	53	30	41
None	Blaze	0	2	1	No	No	No	50	57	65
None	Chaos	7	8	7	No	No	No	57	24	32
Damage	Combo	7	2	1	No	No	No	50	29	29
Damage	Blaze	5	2	2	No	No	No	34	50	57
Damage	Chaos	9	11	8	No	No	No	42	29	33
Partition	Combo	10	13	1	No	Yes	No	39	17	31
Partition	Blaze	0	3	0	No	No	No	41	40	35
Partition	Chaos	12	15	6	No	Yes	No	23	13	36
Power	Combo	10	12	3	Yes	Yes	No	20	19	24
Power	Blaze	5	6	3	No	Yes	No	24	20	29
Power	Chaos	12	14	9	Yes	Yes	Yes	22	16	27

Table 5.1: Experimental Results

presented in Appendix D; "None" means no faults were introduced. The primary damage scenarios are presented in Appendix E. The "Unresolved Crises" column indicates the number of active crises (fires or floods) remaining after the experiment has run for 25 minutes. The "Kill Point" column indicates whether the ship has suffered catastrophic damage (e.g. an explosion in a missile magazine or the loss of all radar systems). The "Overall Score" column reports the score generated by Minerva-5's scoring subsystem and reflects Minerva-5's opinion of how close the results were to the theoretical maximum permitted by the scenario and the induced faults. The scores labeled "M-5" are from a version of Minerva-5.2 configured for shipboard damage control; this system has been deployed as an instructional aid at the U.S. Navy's Surface Warfare Officer School and normally exceeds the performance of human experts on simulated crises [Bul98]. The scores labeled "M-6" are from a "vanilla" version of Minerva-6 configured for shipboard damage control but with none of the fault-tolerance enhancements (zones of jurisdiction,

fault diagnosis, peer monitoring) described in this thesis. Finally, the scores labeled "M-6/FT" are from the enhanced version of Minerva-6 described in this thesis.

Although the Minerva-6 system is still in the early stages of development and does not yet have very refined deliberation or evaluation modules, the Minerva-6/FT extensions make it competitive with the more polished Minerva-5 system. The fault tolerance extensions do seem to give the Minerva-DC system a reasonable chance of saving the ship even when a very catastrophic fault pattern (loss of a machine or network connectivity) occurs, and the Minerva-6/FT system normally does better than even Minerva-5 in those situations. Most importantly, Minerva-6/FT lost only one ship in the twelve test scenarios, compared to five for Minerva-6 and two for Minerva-5. Minerva-5's "Yardstick" scoring mechanism is clearly more biased toward Minerva-5's own behavior than originally anticipated, but it still provides a good relative ranking between Minerva-6 and Minerva-6/FT. The most significant performance differences were in the presence of a network partition, demonstrating the value of the distributed blackboard architecture and the jurisdiction concept. Overall, the Minerva-6/FT architecture seems to be a promising foundation for a production-quality fault tolerant supervisory control system.

Chapter 6

Thesis Contributions and Conclusions

6.1 Contributions

The primary contribution of this thesis is a powerful generic architecture for the construction of distributed fault-tolerant reasoning systems, mating the power of rule-based expert systems and the reliability of commercial distributed databases to enhance the utility of each. The architecture was designed from the ground up to support a wide range of reasoning and fault detection/recovery techniques and allow existing systems to be enhanced with these capabilities without significant modification. The peer monitoring, model-based fault diagnosis, and distributed cooperation algorithms are not novel, but I believe their combination is. Of all the techniques discussed herein, I believe my five-role decomposition of a complex reasoning system into functionally distinct modules that can be easily protected from significant faults is the most important and the most potentially useful to

future efforts.

6.1.1 Theoretical Contributions

- Decomposition of the Minerva deliberation cycle into functionally independent roles, allowing distributed operation, simplified development, and seamless enhancement by features not envisioned in the original design (e.g. fault tolerance, realtime constraints, etc.)

- Use of critiquing facilities to evaluate peer performance and decide when to seize responsibilities from a failed system

- Use of the message-passing paradigm and (Expected Response, Contingency Plan) pairs to diagnose and handle failures in external components or communication channels

- Use of jurisdiction zones to facilitate distributed scheduling with the ability to override failed or slow peers

6.1.2 Practical Contributions

- Low-level blackboard interface to hide communication details from the application

- Configuration of Oracle 8's Multimaster Replication and Conflict Resolution mechanisms to provide the consistent external blackboard required by Minerva-6

- Message-level fault simulator for realistic performance evaluation

- Domain-independent scheduling and plan execution architectures for diagnosing failures and adjusting behavior accordingly
- Domain-specific knowledge-based fault diagnosis model
- Domain-specific evaluation rules for planning around external faults

6.2 Conclusions

6.2.1 Thesis Summary

This thesis describes a highly modular architecture for constructing reliable distributed expert systems, with a focus on deployment in dangerous or unpredictable domains. The effort required to achieve a reasonable level of protection from failure in this architecture is low compared to traditional fault-tolerant systems, primarily due to the relatively flexible view of system state and powerful deduction mechanisms common to blackboard expert systems. The thesis details the design and implementation of the Minerva-6/FT system, with a heavy emphasis on applicability to the shipboard damage control domain, but the techniques described should be useful in a wide variety of expert system architectures and for almost any supervisory control domain. By assuming an abstract view of the complete supervisory control system and adopting specific countermeasures for each kind of fault in the taxonomy of Section 3.1, the overall design is capable of dealing with a wide variety of malfunctions and uncertain inputs, without making it difficult to upgrade specific portions of the rulebase. The modular design also allows different aspects of the system to execute on different servers, monitoring each other for failures and adapting to a dynamic pool of

available resources.

The ultimate goal of the Minerva-6/FT project is to produce a prototype automated damage response system reliable enough to support a reduced personnel contigent aboard future surface warships, so it is critical that these ideas be validated in realistic trials. The results presented in Section 5.2 of this thesis show that Minerva-6/FT performs significantly better than Minerva-6 when faults are introduced into the simulated environment, and even better than the more refined Minerva-5 when the fault pattern (e.g. a network partition) prevents the system from dealing with problems in some areas of the ship without a coordinated distributed response. Although this test took place in a simulated environment, it demonstrates the ability of the Minerva-6/FT architecture to cope with a variety of realistic fault patterns and gives the authors confidence that it can perform reliably in real crisis situations.

6.2.2 Future Research Directions

Two promising ways to improve the performance of the current Minerva/FT system would be the introduction of machine learning in the scheduling modules, allowing them to adjust the system's decisions to reflect known or perceived faults or inadequacies among the participants, and a more sophisticated failure diagnosis model. The current rule base performs adequately but is capable of only coarse fault diagnosis, and the evaluation rules that use its conclusions miss many opportunities to use information about possible failures to the greatest extent. The Time Interval Petri Networks used elsewhere in Minerva-6's evaluation modules seem well suited to modeling failure modes and effects in complex

distributed systems, and would be a likely architecture for future refinement of the failure diagnosis modules. Another interesting area of research would be application of this system architecture to other domains, verifying that it is of practical value as a general supervisory control architecture and determining just how much modification is needed to meet the requirements of other supervisory control problems.

Appendix A

The Minerva-6/FT Domain Action Execution Rules

The following Prolog code is from the domain action execution module of Minerva-6/FT's shipboard damage control configuration. `execute_newly_selected_actions/0` is the primary entry point, responsible for assembling a list of all actions marked as "committed" by the scheduling modules but not yet executed and ultimately passing each to `execute_single_action/0`. `execute_single_action/0` consults `discard_outgoing_message/1` to see if it should be discarded to simulate a faulty network or message target and passes all non-discarded actions to `execute_domain_action/2`, which is responsible for setting up any expected responses the particular class of action (e.g. acknowledgements for orders) and then generating the appropriate control messages for the action. At the bottom level, `send_control_message/3` sets up the list of expected responses for the action according to the `expected_response/3` rules at the end of the appendix.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                                            %%
%% - output.pl                                                                %%
%%                                                                            %%
%% High-level implementation of order execution, with considerations          %%
%% for fault detection and recovery.                                          %%
%%                                                                            %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Core action plan execution engine (domain-independent)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Execute & forget all committed actions that this module can handle
execute_newly_selected_actions :-
        forall(bb_committed_action_table(Id),
               (
                    bb_committed_action_plan(Id, Plan),
                    execute_action_plan(Plan)
                    ;
                    true
               )).


% Attempt to execute an action plan, forgetting it if the execution was successful
execute_action_plan(Plan) :-
        execute_action_chain(Plan),
        (
         Plan = []
         -> true
         ;
         bb_forget_committed_action_plan(Plan)
        ).


% Execute a chain of domain actions
execute_action_chain([]).
execute_action_chain([Car | Cdr]) :-
        execute_single_action(Car),
        execute_action_chain(Cdr).
```

```
% Execute a single domain action
m6_debug_spy_point(execute_single_action/1).
execute_single_action(Action) :-
        bb_hypothesis_head_details(Action, ActionName, Details),
        (
         discard_outgoing_message(Details)
         -> true
         ;
         execute_domain_action(ActionName, Details)
        ),
        pertinent_details(Action, Pattern),
        assert(executed_action(Pattern)).


% Filter out the 'uninteresting' action details
pertinent_details(Action, Pattern) :-
        bb_hypothesis_head_details(Action, Head, D1),
        bb_remove_detail(id, D1, D2),
        bb_remove_detail(timestamp, D2, D3),
        bb_remove_detail(module, D3, D4),
        bb_remove_detail(reason, D4, D5),
        Details = D5,
        Pattern = Head(Details).


% Decide what actions are no longer being executed when a station
% reports no personnel available
detect_action_failure(Station, Problem) :-
        Problem = investigation
         -> forall(no_investigators_for(Station, Action),
                   retract(executed_action(Action)))
        ;
        forall(no_personnel_for(Station, Problem, Action),
               retract(executed_action(Action))).


% Relation between "no personnel available" message details and 'executed'
% actions that were not really executed because the station ran out of personnel
no_personnel_for(Station, Problem, Action) :-
        executed_action(Action),
        bb_hypothesis_head_details(Action, Head, Details),
        bb_extract_detail(who, station, Details),
        bb_extract_detail(problem, Problem, Details),
        not bb_hypothesis(acknowledgment(Details)).
```

```
% Relation between "no personnel available" message details and 'executed'
% investigations that were not really executed because the station ran out of personn
no_investigators_for(Station, Action) :-
        executed_action(Action),
        bb_hypothesis_head_details(Action, Head, Details),
        bb_extract_detail(action, investigate, Details),
        bb_extract_detail(who, station, Details),
        not bb_hypothesis(acknowledgment(Details)).


% Decide to drop an outgoing message to simulate an external fault
discard_outgoing_message(Details) :-
        bb_extract_detail(who, Station, Details),
        ftsim_inaccessible_station(Station, P),
        bb_random(100, X),
        X < P
        ;
        bb_extract_detail(compartment, Comp, Details),
        ftsim_inaccessible_compartment(Comp, P),
        bb_random(100, X),
        X < P.


% Add an "expectation" to the queue
expect(Delay, Constraint, ExceptionHandler) :-
        bb_current_time(ST),
        ET is ST + Delay,
        Goal = (Constraint -> true ; ExceptionHandler),
        bb_cron_job(ET, Goal).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Domain-specific action execution rules
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Attempt to issue an order
execute_domain_action(order, Details) :-
        bb_extract_detail(action, Action, Details),
        determine_message_origin(Details, From),
        (
                bb_extract_detail(who, Station, Details),
                issue_order(Action, Station, Details, From),
```

```
                (
                 From \= dca
                        -> true
                 ;
                 bb_current_time(CT),
                 order_acknowledgement_delay(Action, Station, Details, Delay),
                 expect(Delay,
                        order_acknowledged(Action, Station, Details, CT),
                        no_acknowledgement_received(Action, Station,
                                                    Details, CT))
                )
                ;
                bb_debug_print(['Unhandled order', Action, Details])
        ).


% Attempt to issue a request
execute_domain_action(request, Details) :-
        bb_extract_detail(action, Action, Details),
        determine_message_origin(Details, From),
        (
                bb_extract_detail(who, Station, Details),
                issue_request(Action, Station, Details, From)
                ;
                bb_debug_print(['Unhandled request', Action, Details])
        ).

% Attempt to issue a recommendation
execute_domain_action(recommend, Details) :-
        bb_extract_detail(action, Action, Details),
        determine_message_origin(Details, From),
        (
                bb_extract_detail(who, Station, Details),
                issue_recommendation(Action, Station, Details, From)
                ;
                bb_debug_print(['Unhandled recommendation', Action, Details])
        ).

% Attempt to issue a report
execute_domain_action(announce, Details) :-
        bb_extract_detail(action, Action, Details),
        determine_message_origin(Details, From),
        (
                bb_extract_detail(who, Station, Details),
                issue_report(Action, Station, Details, From)
```

```
                ;
                bb_debug_print(['Unhandled report', Action, Details])
        ).

% Print a warning for all actions not handled by the preceding rules
execute_domain_action(ActionName, Details) :-
        bb_debug_print(['Unsupported domain action', ActionName, Details]).


% Select an appropriate origin for the final control message to be sent
determine_message_origin(Details, From) :-
        bb_extract_detail(inhibit_execution, true, Details)
                -> From = epn
        ;
        From = dca.


% Send the final control message to the appropriate channel
% (i.e. ECLMessages for "real" orders, EPNMessages for critiquing)
send_control_mesg(From, Num, Mesg, Action, Class, ActionName) :-
        From = epn
                -> bb_action_utility(Action, Utility),
                   send_epn_mesg(Num, [num2 = Utility | Mesg])
        ;
        send_ecl_mesg(Num, Mesg),
        prepare_followup(Class, ActionName, Action).


% Set up all "followup" constraints for a domain action
prepare_followup(Class, ActionName, Action) :-
        % Backtrack through all matching expected_response/3 rules
        fault_tolerant,
        expected_response(Class, ActionName, Action),
        fail
        ;
        true.


% Assemble & issue an order to some station
issue_order(Action, To, Details, From) :-
        bb_extract_detail(type, Type, Details),
        order_type_action_name(Type, Action, ActionName),
        order_to_ecl(ActionName, To, Details, From, Num, Mesg),
        RealNum is 1100 + Num,
        send_control_mesg(From, RealNum, Mesg, order(Details),
```

```
                        order, ActionName).


% Assemble & issue a request for information
issue_request(Action, To, Details, From) :-
        bb_extract_detail(type, Type, Details),
        request_type_action_name(Type, Action, ActionName),
        request_to_ecl(ActionName, To, Details, From, Num, Mesg),
        RealNum is 1500 + Num,
        send_control_mesg(From, RealNum, Mesg, request(Details),
                        request, ActionName).


% Assemble & issue a recommendation to some station
issue_recommendation(Action, To, Details, From) :-
        bb_extract_detail(type, Type, Details),
        recommendation_type_action_name(Type, Action, ActionName),
        recommendation_to_ecl(ActionName, To, Details, From, Num, Mesg),
        RealNum is 1600 + Num,
        send_control_mesg(From, RealNum, Mesg, recommend(Details),
                        recommendation, ActionName).


% Assemble & issue an information report
issue_report(Action, To, Details, From) :-
        bb_extract_detail(type, Type, Details),
        report_type_action_name(Type, Action, ActionName),
        report_to_ecl(ActionName, To, Details, From, Num, Mesg),
        RealNum is 1600 + Num,
        send_control_mesg(From, RealNum, Mesg, announce(Details),
                        report, ActionName).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Implementation of order acknowledgement expectation
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Estimate an upper bound on the time to receive an acknowledgement
order_acknowledgement_delay(_Action, _Station, _Details, Delay) :-
        % TODO: improve acknowledgement delay estimation
        Delay = 10.
```

```
% Has an order been acknowledged since it was issued?
order_acknowledged(Action, Station, Details, IssueTime) :-
        % Search through recent acknowledgment hypotheses
        bb_hypothesis(acknowledgement([timestamp = T | Details])),
        T >= IssueTime,
        order_acknowledgement_delay(Action, Station, Details, Max),
        DT is T - IssueTime,
        Max >= DT.


% Is there evidence that some effort was made to execute an order?
order_executed(Details) :-
        bb_hypothesis(success_report(Details))
        ;
        bb_hypothesis(failure_report(Details))
        ;
        bb_hypothesis(in_progress_report(Details))
        ;
        bb_hypothesis(cancellation_acknowledgement(Details)).


% Investigate why no acknowledgment was received for an order
no_acknowledgement_received(_Action, Station, Details, _IssueTime) :-
        % Find failure report or hypothesize that the station is unavailable
        bb_hypothesis(failure_report([timestamp = T | Details])),
        T >= IssueTime
         -> true
        ;
        bb_propose_hypothesis(resource_unavailable([resource=Station, cf=500])).


% Respond to apparent failure to execute an order
order_not_carried_out(Details) :-
        bb_extract_detail(station, Station, Details),
        bb_propose_hypothesis(resource_unavailable([resource = Station,
             cf = 300])),
        bb_propose_hypothesis(anomaly([error = order_not_carried_out,
                                       cf = 600 | Details])).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Expected responses to domain actions
%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Expect that orders are eventually carried out or cancelled
expected_response(order, Type, Action) :-
        order_details(Type, Action, Details),
        expect(180,
               order_executed(Details),
               order_not_carried_out(Details)).

% Expect that the water level will rise in flooded compartments
expected_response(order, flood, Action) :-
        bb_extract_detail(compartment, Compartment, Action),
        expect(120,
               bb_hypothesis(report([type = water_level_rising,
                                     compartment = Compartment])),
               bb_propose_hypothesis(anomaly([error = compartment_not_flooding,
                                              compartment = Compartment,
                                              cf = 800]))).

% Track the boundaries that have already been ordered
expected_response(order, set_boundaries, Action) :-
        bb_extract_detail(boundaries, Boundaries, Action),
        bb_extract_detail(problem, Problem, Action),
        convert_boundaries_internal_to_ecl(Boundaries, SA, PA, PF, SF, A, B),
        assert(current_boundaries(Problem, SA, PA, PF, SF, A, B)),
        expect(120,
               bb_hypothesis(acknowledgement([problem = Problem, boundaries = Boundar
               retract(current_boundaries(Problem, SA, PA, PF, SF, A, B))).

% Expect that requested MR/Z status reports are eventually given
expected_response(request, report_mrz_status, Action) :-
        bb_extract_detail(who, Station, Action),
        bb_current_time(CT),
        expect(60,
               (
                bb_hypothesis(report([type = mrz, who = Station,
                                      timestamp = T])),
                T >= CT
               ),
               bb_propose_hypothesis(anomaly([error = mrz_report_not_given,
                                              who = Station]))).



% Assemble the list of details describing a domain action
```

```
order_details(_Type, Action, Details) :-
        bb_remove_detail(head, Action, Details).
```

Appendix B

The Minerva-6/FT Jurisdiction Assignment and Maintenance Rules

The following Prolog code is from the jurisdiction maintenance phase of the main Minerva-6/FT control loop. The `juris_monitor_peers/0` predicate is periodically invoked by each module to report its continued existence and claimed zones of jurisdiction to accessible machines and to initiate the jurisdiction reassignment protocol for module/jurisdiction combinations which have not recently "reported in".

```
%
% Implementation of the Minerva jurisdiction system.

:- dynamic jurisdiction/1.
:- dynamic juris_owner_alive_fact/2.


% Simulated location of the system, based on initial jurisdiction zone
juris_homebase(none, '1-304-2').         % DC Central
juris_homebase(all, '1-304-2').         % DC Central
juris_homebase(repair_2, '1-97-1').         % Repair-2's locker
```

```
juris_homebase(repair_3, '2-410-2').        % Repair-3's locker
juris_homebase(repair_5, '1-206-3').        % Repair-5's locker


% Relation between zones of jurisdiction and the last heartbeat for each
juris_owner_alive(Zone, T) :-
        juris_owner_alive_fact(Zone, T)
         -> true;
        T = 0.


% How long to wait before even considering seizing a jurisdiction zone
jurisdiction_grace_period(60).


% Max delay between heartbeat messages before jurisdiction is seized
jurisdiction_seize_time(30).


juris_initial_jurisdiction(Zone) :-
        juris_homebase(Zone, Home),
        assert(ftsim_my_location(Home)),
        juris_take_jurisdiction(Zone).

% Call this at startup to make sure some jurisdiction zone is defined.
juris_default_jurisdiction :-
        jurisdiction(_)
        ;
        juris_take_jurisdiction(all).


% Claim a new zone of jurisdiction
juris_take_jurisdiction(Zone) :-
        assert(jurisdiction(Zone)),
        juris_report_jurisdiction(Zone).


% The 'all' zone owns every compartment
jurisdiction_over(Compartment) :-
        jurisdiction(all).


% Compare compartment location with jurisdiction zone
jurisdiction_over(Compartment) :-
        jas(Compartment, Zone),
```

```
        jurisdiction(Zone).


% Look for unclaimed zones and seize them
juris_seize_jurisdiction :-
        bagof(X, juris_seize_jurisdiction(X), _)
        ;
        true.


% Check to make sure a zone is being monitored, otherwise seize it
juris_seize_jurisdiction(Zone) :-
        jurisdiction(Zone)
        ;
        bb_current_time(MT),
        jurisdiction_grace_period(GP),
        MT > GP,
        juris_owner_alive(Zone, AT),
        bb_guard(Delay is MT - AT),
        jurisdiction_seize_time(ST),
        Delay > ST,

        write('*** SEIZING JURISDICTION ZONE '),
        write(Zone),
        write(' ***'),
        nl,

        juris_take_jurisdiction(Zone).


% Send heartbeat messages
juris_report_jurisdiction :-
        bagof(X, juris_report_jurisdiction(X), _)
        ;
        true.


% Every jurisdiction_seize_time / 2 seconds, send the heartbeat message
juris_report_jurisdiction(Zone) :-
        jurisdiction(Zone),
        bb_current_time(MT),
        juris_owner_alive(Zone, AT),
        bb_guard(Delay is 2 * (MT - AT)),
        jurisdiction_seize_time(ST),
        Delay > ST
```

```
         -> retractall(juris_owner_alive_fact(Zone, _)),
            assert(juris_owner_alive_fact(Zone, MT)),
            bb_propose_hypothesis(jurisdiction_over([zone = Zone]))
        ;
        true.


% Handle incoming heartbeat messages
juris_receive_jurisdiction_report(Zone) :-
        bb_current_time(MT),
        retractall(juris_owner_alive_fact(Zone, _)),
        assert(juris_owner_alive_fact(Zone, MT)).


% Fault-tolerance stuff
juris_monitor_peers :-
        juris_report_jurisdiction,
        juris_seize_jurisdiction.
```

Appendix C

An Evaluation Module for Fault-Tolerance Considerations

The following Prolog code is from a rule-based evaluation module used to score proposed action plans in accordance with detected faults. For example, this evaluator will score a plan down if it relies on a resource believed to be malfunctioning. The most interesting rules are the `rbeval_rule_applies/2` and `rbeval_rule_score/3` rules, which determine which scoring rules apply to a particular action and what the effect of each should be in a particular situation.

```
% rbeval.pl -
%
%     Framework for simple rule-based evaluation.  Used primarily to
%     encode scheduling knowledge for "working around" system faults.


:- current_predicate(fault_tolerant/0) -> true ; dynamic(fault_tolerant/0).
:- dynamic executed_action/1.
:- dynamic current_boundaries/7.
```

```
% Invoke the rule-based evaluator on all scorable action plans
% m6_debug_spy_point(rbeval_run/0).
rbeval_run :-
        forall(bb_debatable_action_table(Plan), rbeval_score_action(Plan)).


% Run the rule-based evaluator on a particular action plan
rbeval_score_action(ActionId) :-
        bagof(Name, rbeval_rule_applies(Name, ActionId), Rules),
        rbeval_combine_scores(Rules, ActionId, Score),
        (Score = [] -> true ; bb_post_score_from_id(ActionId, Score))
         -> true
        ;
        true.


% Merge a list of score changes into an existing list of scores
rbeval_merge_score_vectors([], OldScores, OldScores).
rbeval_merge_score_vectors([Car | Cdr], OldScores, NewScores) :-
        rbeval_insert_score(Car, OldScores, Scores),
        rbeval_merge_score_vectors(Cdr, Scores, NewScores).


% Merge a single score change into an existing list of scores
rbeval_insert_score(Score, [], [Score]).
rbeval_insert_score(Score, [Car | Cdr], AdjustedScores) :-
        Score =.. [=, Head, V1],
        Car =.. [=, Head, V2]
         -> bb_guard(Sum is V1 + V2),
            NewScore =.. [=, Head, Sum],
            AdjustedScores = [NewScore | Cdr]
        ;
        rbeval_insert_score(Score, Cdr, NewScores),
        AdjustedScores = [Car | NewScores].


% rbeval_combine_scores/3 is shorthand for rbeval_combine_scores/4 with an
% empty initial score vector
rbeval_combine_scores(Rules, ActionId, Scores) :-
        rbeval_combine_scores(Rules, ActionId, [], Scores).

% rebeval_combine_scores/4 succeeds when the last argument unifies with
% the third argument adjusted by the total result of the first two arguments
```

```
rbeval_combine_scores([], _ActionId, Scores, Scores).
rbeval_combine_scores([Car | Cdr], ActionId, OldScores, FinalScores) :-
        (rbeval_rule_score(Car, ActionId, Scores) -> true; Scores = []),
        rbeval_merge_score_vectors(Scores, OldScores, NewScores),
        rbeval_combine_scores(Cdr, ActionId, NewScores, FinalScores).


% rbeval_rule_applies/2 succeeds when a rule is relevant to the given action
rbeval_rule_applies(redundant_action, ActionId) :-
        bb_action(ActionId, Action),
        pertinent_details(Action, Pattern),
        executed_action(Pattern).

rbeval_rule_applies(fight_problem, ActionId) :-
        bb_action(ActionId, [head = order, action = fight]).

rbeval_rule_applies(set_boundaries, ActionId) :-
        bb_action(ActionId, [head = order, action = set_boundaries]).

rbeval_rule_applies(investigate, ActionId) :-
        bb_action(ActionId, [head = order, action = investigate]).

rbeval_rule_applies(flood_compartment, ActionId) :-
        bb_action(ActionId, [head = order, action = flood]).

rbeval_rule_applies(out_of_jurisdiction, ActionId) :-
        fault_tolerant,
        bb_action(ActionId, [compartment = Comp]),
        not jurisdiction_over(Comp).

rbeval_rule_applies(resource_unavailable, ActionId) :-
        fault_tolerant,
        bb_action(ActionId, [who = Target]),
        bb_hypothesis(resource_unavailable([resource = Target])).


% rbeval_rule_score/3 represents the relation between rules, actions, and
% the scoring adjustment for that action according to that rule

rbeval_rule_score(redundant_action, _ActionId, Score) :-
        % TODO: change to degrade penalty by time since original action
        Score = [raw = -1000].

rbeval_rule_score(fight_problem, ActionId, Score) :-
        bb_action(ActionId, [compartment = Comp, problem = Problem]),
```

```
        bb_hypothesis(crisis([type = Problem, compartment = Comp, cf = CF])),
        (rbeval_problem_weight(Problem, Comp, Weight) -> true; Weight = 0),
        bb_guard(RawScore is CF * Weight),
        Score = [raw = RawScore].

rbeval_rule_score(set_boundaries, ActionId, Score) :-
        bb_action(ActionId, [boundaries = Boundaries, problem = Problem]),
        (
         rbeval_redundant_boundaries(Problem, Boundaries)
         -> Score = [raw = -100]
         ;
         rbeval_boundary_coverage(Boundaries, Coverage),
         (rbeval_boundary_weight(Problem, Weight, Base) -> true ; Weight = 0, Base =
         bb_guard(RawScore is Base + Coverage * Weight),
         Score = [raw = RawScore]
        ).

rbeval_rule_score(investigate, _ActionId, Score) :-
        (
         important_compartment(Comp)
         -> bb_guard(RawScore is 100)
         ;
         bb_guard(RawScore is -50)
        ),
        Score = [raw = RawScore].

rbeval_rule_score(flood_compartment, _ActionId, Score) :-
        bb_action(ActionId, [compartment = Comp]),
        bb_hypothesis(crisis([compartment = Comp, cf = CF])),
        magazine_compartment(Comp),
        bb_guard(RawScore is 2 * CF),
        Score = [raw = RawScore].

rbeval_rule_score(out_of_jurisdiction, _ActionId, Score) :-
        Score = [raw = -800].

rbeval_rule_score(resource_unavailable, ActionId, Score) :-
        bb_action(ActionId, [who = Target]),
        bb_hypothesis(resource_unavailable([resource = Target, cf = CF])),
        bb_guard(RawScore is CF / -2),
        Score = [raw = RawScore].


% Are the recommended boundaries redundant?
rbeval_redundant_boundaries(Problem, Boundaries) :-
```

```
        convert_boundaries_internal_to_ecl(Boundaries, SA, PA, PF, SF, A, B),
        current_boundaries(Problem, SA2, PA2, PF2, SF2, A2, B2),
%        SA2 >= SA,
        PA2 >= PA,
        PF >= PF2,
%        SF >= SF2,
        A >= A2,
        B2 >= B.

% Compute a rough value of the worth of boundaries this size
rbeval_boundary_coverage(Boundaries, Coverage) :-
        convert_boundaries_internal_to_ecl(Boundaries, SA, PA, PF, SF, A, B),
        bb_guard(PCoverage is PA - PF),
        bb_guard(SCoverage is PA - PF),
        bb_guard(Decks is B - A),
        bb_guard(Coverage is PCoverage * Decks - SCoverage).


% Relation between the types of boundaries and their respective utilities
rbeval_boundary_weight(fire, 0.25, 100).
rbeval_boundary_weight(flood, 0.1, 0).
rbeval_boundary_weight(smoke, 0, -100).


% Relation between types of crises and their relative threat
rbeval_problem_weight(fire, Comp, Weight) :-
        important_compartment(Comp)
         -> Weight = 1.0
        ;
        Weight = 0.33.

rbeval_problem_weight(flood, _Comp, 0.15).
rbeval_problem_weight(smoke, _Comp, -0.25).
```

Appendix D

Fault Patterns Used in the Experiments

These files contain the fault patterns used in the experimental evaluation of the Minerva-6/FT architecture. The effects of each predicate are described in the comments at the top of each file.

```
% chaos.pl -
%
%   Combine all of the other fault patterns.

ftsim_load_fault_pattern(damage).
ftsim_load_fault_pattern(partition).
ftsim_load_fault_pattern(power).

% faults/damage.pl -
%
%   Simulation of extensive damage to external systems (personnel, sensors,
% and actuators) on an Arleigh Burke-class destroyer.

% Syntax:
%
```

```
% ftsim_lose_station(Frame, Time) disables communication with the
% specified personnel station or automated system after the given time.
%
% ftsim_lose_frames(Direction, Frame, Time) disables communication with areas
% of the ship fore or aft of the given frame after the specified time.
%
% ftsim_lose_deck(Deck, Time) disables communication with the given deck of the
% ship after the specified time.
%
% All of these predicates take an optional third parameter giving the message
% loss rate (on a scale of 0-100) to simulate intermittent network or component
% faults.
%

% Simulate 70% failure rate in the temperature alarm system
ftsim_lose_station(temperature_alarm, 30, 70).

% Simulate 50% failure rate aft of frame 174
ftsim_lose_frames(aft, 174, 90, 50).

% Simulate complete loss of Repair Locker 5
ftsim_lose_station(repair_5, 20).

% Simulate minor disruption (10%) of all communication throughout the ship
ftsim_lose_frames(aft, 0, 60, 10).
```

```
% faults/partition.pl -
%
%   Simulation of a network partition at frame 174 of an Arleigh Burke-class
% destroyer.

% Syntax: ftsim_partition_frame(Frame, Time) causes the input & output modules
% to discard all traffic crossing the specified frame after the given time,
% including control messages within external systems (e.g. alarm circuits
% between sensors and alarm panels)

ftsim_partition_frame(174, 60).
```

```
% faults/power.pl -
%
%   Simulation of loss of electrical power to many areas of an
% Arleigh Burke-class destroyer.

% Simulate loss of the temperature alarm system
ftsim_lose_station(temperature_alarm, 30, 70).

% Simulate loss of four firemain pumps
ftsim_lose_pump(firemain(1), 30).
ftsim_lose_pump(firemain(2), 30).
ftsim_lose_pump(firemain(3), 30).
ftsim_lose_pump(firemain(4), 30).

% Simulate loss of communications & power to CIC and the bridge
ftsim_lose_compartment('1-126-0', 120).
```

Appendix E

Primary Damage Scenarios Used in the Experiments

These files are the primary damage specifications given to the simulator for each of the scenarios used in the experimental evaluation of Minerva-6/FT. The first value in each row is the number of seconds into the scenario at which the event occurs. The second value in each row is the type of event to simulate, either ignition of all combustibles in a compartment or rupture of piping and/or the ship's hull in a compartment. The third value on each row is the compartment number associated with the compartment, which consists of the compartment's deck, frame, and position separated by hyphens. The deck number indicates the distance (in decks) from this compartment to the waterline, with negative decks being above the waterline and deck 1 being right at the waterline (there is no deck 0). The frame number indicates the distance in feet from the bow of the ship to the forward-most wall of the compartment. Finally, the position number indicates the compartment's position starboard or port of the ship's centerline, with odd numbers indicating starboard

compartments and even numbers indicating port compartments.

```
Scenario Scripter v2.0
Combo 1

# Multiple small fires, flooding in Engineering spaces

25        Flood 4-126-0
30        Flood 4-220-0
35        Flood 4-220-1
40        Flood 3-220-100
45        Ignite 3-338-1
1:15        Ignite 3-370-0
1:30        Ignite 1-268-2
2:00        Ignite 1-78-1
2:35        Ignite 2-42-1
2:40        Ignite 2-174-4
2:50        Ignite 2-18-0
3:10        Flood 5-174-1
3:15        Flood 4-174-1
3:30        Ignite 4-42-0
3:45        Ignite 1-54-1
4:00        Flood 4-418-1
7:00        Ignite 1-46-0

Scenario Scripter v2.0
Combo 8

# Fires and flooding rapidly appear throughout the ship

45        Ignite 1-186-1
55        Ignite 2-53-1
1:00        Ignite 1-18-0
1:10        Flood 3-220-100
1:15        Flood 4-254-0
1:20        Flood 4-126-0
1:30        Ignite 2-42-1
1:45        Ignite 3-97-2
2:05        Ignite 2-418-1
2:15        Ignite 2-174-4
2:30        Flood 4-410-2
2:35        Ignite 1-54-2
2:45        Flood 4-42-0
```

```
3:00        Ignite 1-126-0
3:05        Ignite 1-84-1
3:15        Ignite 2-126-1
3:25        Ignite 4-94-0
3:40        Ignite 1-100-2
3:55        Ignite 1-97-1
4:10        Ignite -1-294-2



Scenario Scripter v2.0
Difficult Fire 3

# Medium-size fire in the ship's superstructure

1:00        Ignite -1-110-0
1:35        Ignite -1-122-2
2:40        Ignite -1-274-1
3:00        Ignite -3-142-2
4:30        Ignite -4-272-0
```

Bibliography

[Ack84] J. Ackermann. Robustness against sensor failures. *Automatica*, 20(2):211–215, 1984.

[Ack87] J. Ackermann. Challenges to control: a collective view. *IEEE Transactions on Automatic Control*, 32(4):275–285, 1987.

[AFSV98] G. Arroyo-Figueroa, E. Solis, and A. Villavicencio. Sadep–a fuzzy diagnostic system shell–an application to fossil power plant operation. *Expert Systems with Applications*, 14:43–52, 1998.

[BJ87] K.P. Birman and T.A. Joesph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[Bul97] Vadim Bulitko. Minerva 4. CS444 Class Project Presentation, May 1997.

[Bul98] Vadim Bulitko. Minerva-5: a multifunctional dynamic expert system. Master's thesis, University of Illinois at Urbana-Champaign, May 1998.

[CT96] Deepak Tushar Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[Fra90] P.M. Frank. Fault diagnosis in dynamic systems using analytical and knowledge-based redundancy– a survey and some new results. *Automatica*, 26:459–474, 1990.

[FYR98] Enbo Feng, Haibin Yang, and Ming Rao. Fuzzy expert system for real-time process condition monitoring and incident prevention. *Expert Systems with Applications*, 15:383–390, 1998.

[GAMdP98] C. Alonso Gonzalez, G. Acosta, J. Mira, and C. de Prada. Knowledge based process control supervision and diagnosis: the aerolid approach. *Expert Systems with Applications*, 14:371–383, 1998.

[Ise84] R. Isermann. Process fault detection based on modeling and estimation methods– a survey. *Automatica*, 20(4):459–474, 1984.

[KBE97] E. Kaszkurewicz, A. Baya, and N. F. F. Ebecken. A fault detection and diagnosis module for oil production plants in offshore platforms. *Expert Systems with Applications*, 12(2):189–194, 1997.

[Kim95] Young-Jin Kim. A framework for an on-line diagnostic expert system for intelligent manufacturing. *Expert Systems with Applications*, 9(1):55–61, 1995.

[Liu96] Wei Liu. An on-line expert system-based fault-tolerant control. *Expert Systems with Applications*, 11(1), 1996.

[MZG+95] J. Mendigutxia, P. Zubizarreta, J. M. Goenaga, L. Berasategui, and L. Manero. Fault tolerance in automated manufacturing systems. *Expert Systems with Applications*, 8(2):275–285, 1995.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verifications for fault-tolerant architectures: Prologomena to the design of pvs. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[PDK95] M.A. Peraldi, J.D. Decotignie, and T. Kouthon. Intelligent control for safety-critical applications. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, volume 4, pages 2994–2999. IEEE, 1995.

[Pet82] Gary L. Peterson. An $o(n \log n)$ unidirectional distributed algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, 4(4):758–762, 1982.

[PGG+98] Barney Pell, Ed Gamble, Erann Gat, Ron Keesing, Jim Kurien, Bill Millar, P. Pandurang Nayak, Christian Plaunt, and Brian Williams. A hybrid procedural/deductive executive for autonomous spacecraft. In *Proceedings of the Second International Conference on Autonomous Agents*. ACM Press, 1998.

[PTDW92] Y.T. Park, K. W. Tan, S. Donoho, and D. C. Wilkins. Minerva 3.0: A knowledge-based expert system shell with declarative representation and flexible control. Technical Report UIUC-BI-KBS-92-012, University of Illinois at Urbana-Champaign, January 1992.

[PTW91] Y.T. Park, K. W. Tan, and D. C. Wilkins. Minerva: A knowledge-based expert system shell with declarative representation and flexible control.

Technical Report UIUC-BI-KBS-90-017, University of Illinois at Urbana-Champaign, November 1991.

[She98] Anne Sheperd. Knowledge-based expert systems: Critiquing versus conventional approaches. *Expert Systems with Applications*, 14:433–441, 1998.

[Sin97] P. K. Sinha. *Distributed Operating Systems: Concepts and Design.* IEEE Computer Society Press, Piscataway, NJ, 1997.

[SS97] H. Silberschatz, A. Korth and S. Sudarshan. *Database System Concepts.* McGraw-Hill Companies, Inc., Boston, MA, 3rd edition, 1997.

[SS98] Daniel P. Siewiorek and Robert S. Swarz. *Reliable Computer Systems: Design and Evaluation.* A K Peters, third edition, 1998.

[Ste91] R. F. Stengel. Intelligent failure-tolerant control. *IEEE Control Systems Magazine*, 11(4):14–23, 1991.

[WS97] David C. Wilkins and Janet A. Sniezek. An approach to automated situation awareness for ship damage control. Technical Report UIUC-BI-KBS-97012, University of Illinois at Urbana-Champaign, July 1997.