A FAULT-TOLERANT INTELLIGENT ROBOTIC CONTROL SYSTEM

Neville I. Marzwell
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

Kam Sing Tso
SoHaR Incorporated
Beverly Hills, CA 90211

ABSTRACT

This paper describes the concept, design, and features of a fault-tolerant intelligent robotic control system being developed for space and commercial applications that require high dependability. The comprehensive strategy integrates system level hardware/software fault tolerance with task level handling of uncertainties and unexpected events for robotic control. The underlying architecture for system level fault tolerance is the distributed recovery block which protects against application software, system software, hardware, and network failures. Task level fault tolerance provisions are implemented in a knowledge-based system which utilizes advanced automation techniques such as rule-based and model-based reasoning to monitor, diagnose, and recover from unexpected events. The two level design provides tolerance of two or more faults occurring serially at any level of command, control, sensing, or actuation. The potential benefits of such a fault tolerant robotic control system include: 1) a minimized potential for damage to humans, the work site, and the robot itself, 2) continuous operation with a minimum of uncommanded motion in the presence of failures, 3) more reliable autonomous operation providing increased efficiency in the execution of robotic tasks and decreased demand on human operators for controlling and monitoring the robotic servicing routines.

INTRODUCTION

The reliability issue must be addressed before robotic systems can be dependably used in critical applications such as servicing the Space Station Freedom and patient monitoring and tending tasks in medical facilities. This paper describes a comprehensive approach which integrates the handling of hardware, software, communication, and operational errors in robotic systems. Although some work has been done on the handling of uncertainties and unexpected events during task execution [1, 2, 3, 4], there has been little research on the handling of system level hardware and software failures in robotics. The research addresses this void by means of a comprehensive strategy for integrating system level hardware/software fault tolerance with task level handling of uncertainties and unexpected events.

Table 1 shows our integrated approach to robotics fault tolerance. Errors are handled on two levels: the *system level* which includes the computers and other hardware, control software, and communications, and the *task level* which includes anomalies and uncertainties associated with the physical environment during task execution. Examples of faults, errors and recovery, together with the general fault tolerance strategy and our specific approach to handling them are shown for each of these levels.

In our terminology a *failure* is a difference between the actual behavior of a system and the expected behavior. An *error* is an undesired system state and a *fault* can be considered as a low-level failure of some subsystem. In other words, a fault causes the system to get into an error state and the failure behavior is a manifestation of the error state. For example, a faulty motor caused a servo gripper stuck at an erroneous open position which led to the failure to grasp an object.

There are four main classes of errors that can be identified in a robotic control system. These are hardware errors, software errors, communications errors, and operational errors [5].

- *Hardware errors* occur in all kinds of mechanical and electrical mechanisms, in control systems, in sensory devices, and in electronic and computer systems. They are caused either by component failure

Table 1: Integrated Approach to Robotics Fault Tolerance

Level	Class	General Strategy	Example Fault	Example Error	Example Recovery	Specific Approach
System	hardware	redundancy	processor stop	missed output	replaced by backup processor	extended distributed recovery block architecture
	software	design diversity	unanticipated singularity not handled	erroneous setpoint output	assertion check & alternate algorithm	
	comm-unication	coding	line noise	data scrambled	data encoding check & retransmission	
Task	operation	intelligence	weak grip force	object slipped	replan & regrasp	knowledge-based system

or by design faults. A common technique for tolerating hardware failures and faults is the introduction of some form of redundancy. Key components of the system are replicated and work in parallel. If one of the replicated components fails, the remaining components continue to operate. The user does not notice the error, and the system continues to function correctly. The EDRB architecture described in this paper incorporates hardware fault tolerance in the form of a node pair and the associated fault detection and recovery software.

- *Software errors* occur through design faults in programs. With the increase in sophistication of robotic systems, software has become more significant and complex. The conventional technique for software reliability is extensive verification and validation. It is well known that software testing can only reveal the presence of faults, but not their absence. As a result, software fault tolerance techniques have been used to achieve high reliability for critical applications which may endanger human life or entail great financial loss. Tolerance to design faults relies on the application of design diversity, which creates diverse software components from a common requirement. Their diversity is introduced by the use of independent programmers, algorithms, programming languages, and tools. The goal is to increase the probability that software errors will be tolerated by diverse software components. The EDRB architecture incorporates software fault tolerance by using two diverse versions of the software coupled with an on-line acceptance test which can detect failures in either version prior to transmitting their output to the actuators.

- *Communication errors* occur in command and status information communicated between control computers, robotic controllers, and sensory devices. They are caused by transmission error due to noise, loss of synchronization due to timing errors, or loss of data due to hardware failures. The first two failures can be detected by the use of coding and recovered by retransmission. Hardware failures can only be tolerated by redundant communication links. The EDRB architecture addresses communication errors through encoding and redundant communication links.

- *Operational errors* are the physical errors that occur in the robot task environment. These are not software or hardware errors but refer to a range of faults due to uncertainties and unexpected events that happen during task execution. For example, an autonomous robot vehicle might unexpectedly find an obstacle in its path. Alternately, a robot might find that it has failed to grasp an object either because the object is not present or because the object slipped from its grip. Some failures due to defective components are also classified as operational errors because the conventional redundancy technique to tolerate their failures is not viable for them. For example, a "standby robot," even if economically possible, could not access all the operating space of a failed robot and therefore cannot be used to replace the failed robot. Operational errors are the types of error conditions that an intelligent robot must be designed to detect and recover from.

The next sections describe the system level fault tolerance provisions which tolerate hardware, software and communications faults; and the task level fault tolerance provisions which tolerate operational faults.

SYSTEM LEVEL FAULT TOLERANCE

A real-time fault-tolerant distributed architecture called the Extended Distributed Recovery Block (EDRB) [6] will be used to handle system level faults. The underlying fault tolerance algorithms and mechanisms are based on extensions to the distributed recovery block [7] which is in turn based on the classical recovery block [8] with real time extensions.

Figure 1 is a top level diagram of a robotic control system which incorporates the EDRB. This configuration is a typical teleoperated-autonomous dual-arm robotic system with supervised autonomy for space telerobotics [9]. Fault tolerance for hardware, software, and communications failures is provided for the Task Execution System because it is remotely located and must respond rapidly to these failures. Although other system elements are not shown as requiring fault tolerance in this example, nothing precludes the application of the EDRB for the Task Planning System, the interface, or other elements if such were required.

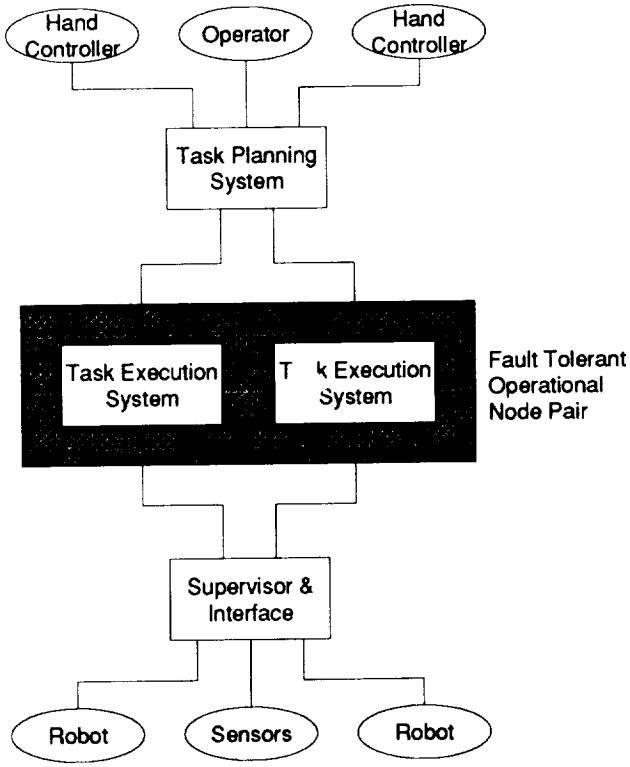


Figure 1: Fault-Tolerant Robotic Control System Based on the EDRB Architecture

In the general terminology of the EDRB, the replicated Task Execution System computers are collectively referred to as an operational node pair. One member of the node pair, called the *active* node, provides active control and processing for the robot and sensors. The other node, referred to as the *shadow*, operates as a standby. The active and shadow nodes exchange frequent periodic status messages, called heartbeats, over redundant communication lines as an indication of their states of health. If the shadow node senses the absence of its companion active node's heartbeat, it will promote itself to the active status after verifying concurrence with a *supervisor*. This concurrence is required in order to prevent a spurious takeover due to faulty communications in the shadow node or a false alarm due to a transient anomaly. After taking over, the newly promoted active node will induce a hardware reset and software reload of the failed node in the hope of restoring it to backup status. The supervisor itself need not be replicated because it is needed only

to assist in recovery; the EDRB can function in steady state without the supervisor.

Figure 2 shows how distributed recovery blocks are implemented in the EDRB. Within both the active and shadow nodes are two versions of the task execution software, referred to as the *primary* and *alternate* routines. Under normal circumstances, the primary routine is run on the active node while the alternate routine is concurrently run on the shadow. The primary routine is coded to provide the greatest functionality, accuracy, and performance. The alternate routine provides less functionality and performance, but is coded to optimize reliability. For example, in a sensor processing application, the primary routine might use Kalman filtering whereas the alternate routine might use a moving average. After each processing iteration, an online acceptance test checks the validity of the output of both the primary and alternate routines. If the acceptance test shows that an error has occurred in the primary routine, the output will be taken from the alternate routine and control is passed to the shadow node.

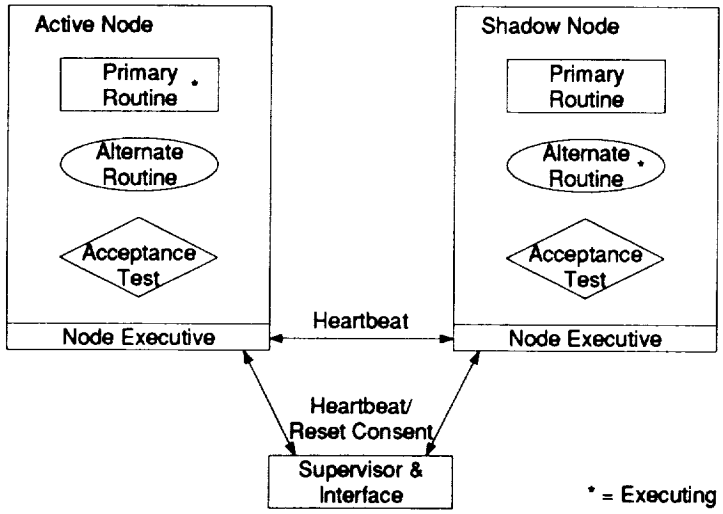


Figure 2: EDRB Software Structure

The EDRB tolerates a broad range of hardware, system software, and application failures including:

- Robotic task execution software not outputting a correct setpoint by the required deadline (by means of acceptance tests, timers, and alternate routines).
- Hardware or system software failures (by means of information encoding, timers, and redundancy)
- Communications link failures (by means of encoding, retransmission, and redundant communication links)
- Spurious recovery actions (by means of the supervisor and consideration of failure histories in the node executive).

One of the most important characteristics of the EDRB for robotic control applications is its fast response and recovery times. The algorithms used in the EDRB fault detection and recovery modules are fast because they do not require any kind of rollback. This characteristic is achieved by executing the primary and alternate routines in parallel.

The EDRB provides the general framework of the primary routine, alternate routine, and acceptance test which work together to tolerate software faults. However, it is necessary to define application specific algorithms for the primary and alternate routines, as well as to define acceptance tests which dependably distinguish between correct and incorrect output.

The diversity to be achieved in the primary and alternate routines is highly dependent upon the application. The free motion of a 7-DOF redundant arm is used to illustrate how software diversity can be

achieved. Two possible independent approaches to *configuration control* of redundant manipulators [10]: (1) the *Jacobian pseudoinverse* [11] which has good tracking but cannot handle singularity, (2) the *damped least square* [12] which is singularity robust but has bad tracking near singularity. The primary routine will use Jacobian pseudoinverse to ensure good tracking whereas the alternate routine will be based on the damped least square when the primary fails to handle singularity. Because many of the software failures in these routines are likely to be in the mathematical operations, the alternate routine will rely on lookup tables instead of math library functions provided by the compiler. On the other hand much of the "framework" coding (i.e., preparation of input, buffering of output, etc.) will be common to both modules because of the lower likelihood of failures. Should experience demonstrate that this is not the case, then these and other software components can also be made diverse.

The acceptance test is the single most critical element of the EDRB. If it fails to reject an incorrect result, or fails to accept a correct result, it comprises a single point of failure. As such, the acceptance test must be both simple and general. While this is a rigorous requirement, it is not impossible to meet in the context of robotic applications. In the free motion example, the acceptance test will determine 1) that the next setpoint is closer to the destination than the previous, 2) the difference between the observed joint angles and the command joint angles are small, 3) the command joint angles are not close to joint limits, and 4) the observed force/torque values are close to the gravitational force of the grasped object.

TASK LEVEL FAULT TOLERANCE

The task level fault tolerance in the proposed design is a knowledge-based system which uses rule-based and model-based reasoning to monitor, diagnose, and recover from unexpected events that occur during the execution of robot tasks.

Most of the present robotic systems handle unexpected events by preprogramming error detection and recovery procedures for every probable error that can be perceived [13]. This approach is inefficient, and it is difficult to completely handle all failures. On the other hand, most of the artificial intelligence research efforts have focused on detection and recovery from failures in simulated robots. They made unrealistic assumptions about the real world and ignored performance and integration issues [3]. Other attempts at automatic error recovery without human intervention have not been used in real applications, because they could not handle the vast range of potential error conditions [2].

The approach outlined in this research addresses these problems as follows:

1. *Emphasis on the support of the robotic task execution system*:

   Most AI research on robotics has emphasized the task planning level. Experience from the NASA/JPL Space Telerobotics Program has shown that monitoring and recovery at the task execution level is both necessary and effective because of its quick response. Our design partitions the fault tolerance strategies into two levels: the *local level* which resides in the task execution system, and the *global level* which resides in the task planning system. The local level provides quick and simple monitoring and recovery actions, while the global level provides extensive and complete monitoring and recovery. The two levels complement each other in their efforts to monitor, diagnose, and recover from failures.

2. *Emphasis on the role of the operator in failure recovery*:

   It is doubtful that any strategy developed for automatic error recovery in a robotic control system can cover all potential failures. Even if such a strategy were developed, it would take some time before confidence in the automatic recovery capabilities would be gained. Therefore, it is necessary to develop a system in which the operator is integrated into the failure recovery process. The operator will always have the capability to approve, query, and intervene a recovery plan. Pertinent information is relayed to the operator with an emphasis on the human/computer interface design.

3. *Emphasis on the performance and integration of the knowledge-based system*:

   Most AI research has been done with familiar AI languages such as Lisp and Prolog in simulated application environments. Problems such as slow response time, communication difficulties, and interface incompatibilities were not addressed. In this research we use C and the CLIPS expert system shell

(implemented in C) to enhance performance and minimize integration problems with the underlying UNIX operating system and the X Window based graphical user interface.

The following subsections describe the local and global task level fault tolerance strategies.

Local Fault Tolerance Strategies

Experience from the NASA/JPL Space Telerobotics Program has shown that local monitoring and recovery actions in the task execution system are both necessary and effective [9]. They are necessary because quick response time is always needed in emergency situations. At the task execution level, monitoring and recovery can be achieved in real time, e.g. at every sample at a rate of 200 Hz. Recovery actions can be initiated at the time of failure occurrence. This is especially crucial in ground/remote telerobot systems where the task planner is located at the ground station and the time delay is significant. Local monitoring and recovery are effective because most failures manifest themselves in excessive force, jerks, or undesired motion. Failures can be detected by monitoring the force/torque thresholds, joint velocities and limits without considering the robotic task context. The recovery action implemented in the JPL Telerobot System was to simply stop the arm, thereby preventing it from damaging the work site and itself. It was found to be an effective initial step for further recovery actions by the operators.

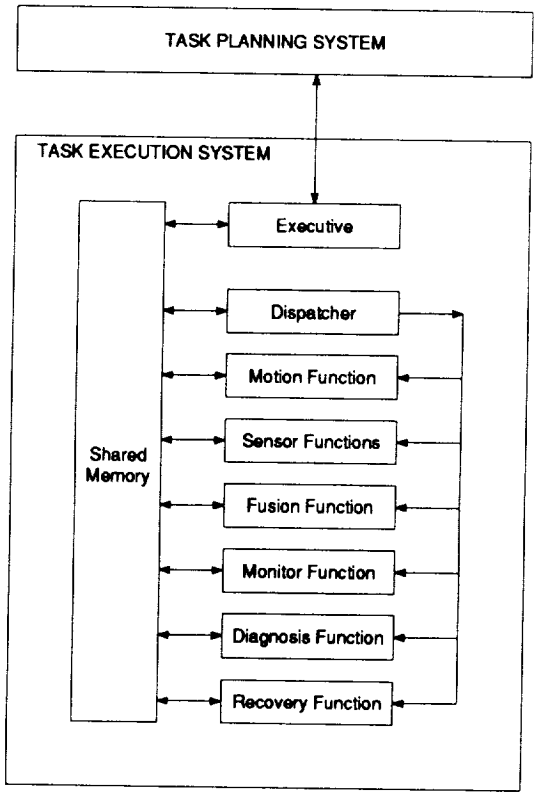


Figure 3: Task Execution System Architecture Supporting Local Fault Tolerance

Local monitoring and recovery are implemented in a task execution system such as the modular architecture [14] as shown in Figure 3. The Executive Process communicates with the Task Planning System to accept new commands and to return new statuses. The Execution Process consists of various modules that provide task execution, monitoring, and reflex capabilities. The *Dispatcher* starts and stops the execution of the various functions. The *Motion Function* sets up the kinematic relationships for interpolated motion. The *Fusion Function* combines the motion perturbations from each sensor with the nominal interpolated motion. These motion perturbations are calculated by the *Sensor Functions*.

This architecture is extended to support local task level fault tolerance as follows:

- *Monitoring* is done in the *Monitor Function*, which tests for various sensor values and conditions in every sampling period. Some examples are force/torque values, joint limits, joint singularities, and elapsed time. The Dispatcher is signaled when an anomaly is detected. The monitor rules implemented in the Monitor Function have the following general form:

  if <situation> **ensure** <condition>

  For example, for continuous collision testing, the monitoring rule is:

  if true **ensure** f/t < safety-threshold

  Another example in monitoring grasping is:

  if contact-sensor = CONTACT **ensure** finger-separation $\approx$ object-size

  The monitor rules not only allow us to test thresholds, they provide a means to monitor a sensor execution profile and test events that occur only in specific situations.

- *Recovery* is activated by the Dispatcher once an anomaly has been signaled by the Monitor Function. The objective of recovery at this level is to provide a fast reflex action to protect the arm and the work site. It is only the initial step of the whole recovery process. Although in most cases stopping the arm is appropriate to safeguard the hardware, there are situations where other recovery actions are needed. For example, if unstable conditions occurred during insertion, stopping the arm may still inflict damaging force to both the arm and the object. Other reflex actions such as relax and return to original position will be implemented.

- *Diagnosis* at this level is used to help the global recovery function to test, re-synchronize, and re-initialize sensors. Specific testing procedures will be devised for each sensor to help the Global Recoverer to determine if it has failed. For example, a force/torque sensor can be tested by comparing ten consecutive readings to ensure that the values are not fixed and that they are reasonable. Many sensor failures are due to communications being out of synchronization or in erroneous internal states. Functions that are able to re-synchronize and re-initialize the sensors will be implemented to assist the global error recovery strategy.

Global Fault Tolerance Strategies

Without the world model and required knowledge and the power to reason, local fault tolerance is limited to detecting errors and using simple reflex actions to protect itself and its work site. Global fault tolerance complements local fault tolerance provisions in that it makes extensive use of spatial reasoning, rule-based reasoning, and model-based reasoning to monitor, diagnose, and recover from failures. Figure 4 shows the architecture of the Task Planning System which supports global fault tolerance.

- *Monitor*: The global monitoring uses both rule-based and model-based reasoning to detect errors that cannot be detected at the local level. The rule-based reasoning is similar to that of the local level but is more sophisticated. For example, if the screwdriver does not seat correctly on the screw in bolt turning, it can be detected by comparing the execution force/torque profile with the force/torque signature stored in the knowledge base. The global monitoring also uses model-based reasoning. For example, a task which inspects the surface of a rectangular frame can detect arm motion errors based on the geometric model of the frame stored in the world model.

- *Diagnosis*: The global diagnosis decides what really occurred based on the raw data indicating an error. For example, a failed grasp may be caused by misorientation of a part/tool, a missing part, slippage of a part, a gripper that cannot close, incorrect compliance, collision, etc. Rules have been developed to perform the diagnosis. These rules use raw sensor data, the semantics and context of the failed task, and the physical behavior of the objects in the work site.

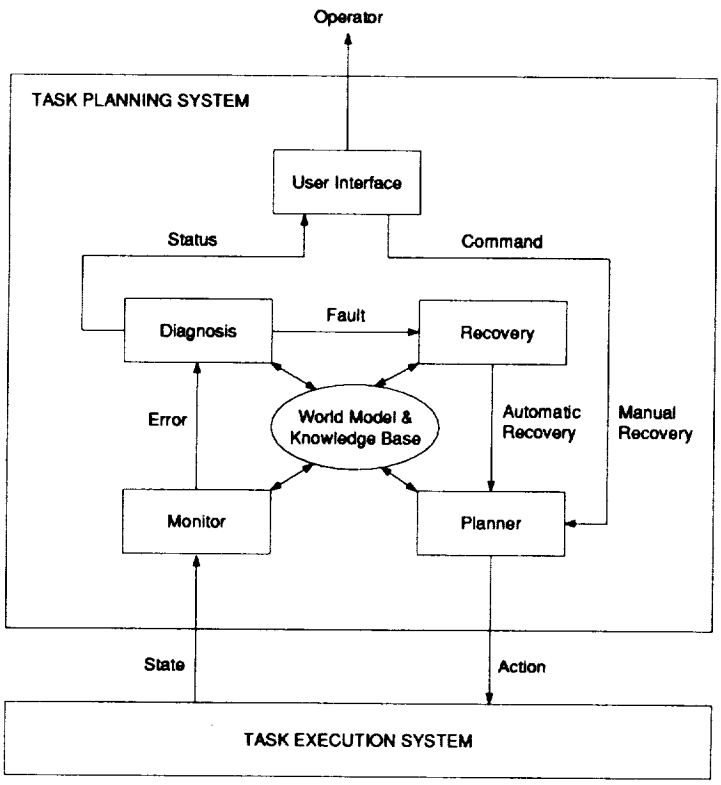


Figure 4: Task Planning System Architecture Supporting Global Fault Tolerance

- *Recovery*: After a fault has been identified, the global recoverer generates pertinent recovery actions according to the fault and the task context. Recovery actions can range from simple to complicated:
  - *Retry*: Faulty readings may be transient and a simple retry of the unfinished plan may be adequate.
  - *New Parameters*: Default parameter values in the task plan may not be appropriate for the situation; new parameter values are used to retry the failed action. An example is unstable compliant motion that becomes stable after the gain is reduced.
  - *Corrective Actions*: Extra actions are needed to correct the erroneous state. For example, a regrasp action is needed after an object slips during grasping.
  - *World Model Update*: An update to the world model is needed because it is found inconsistent with reality. For example, a missing object found during grasping should delete that object from the world model.
  - *Replan*: The original task plan has to be replanned. For example, a new path is used to avoid a collision.
  - *Reconfiguration*: Configuration of the available resources needs to be updated after the hardware has been found to have failed. For example, if a robot arm has failed, the planner should use the other arm to perform its tasks, if possible.
- *Planner*: The planner stores task plans for nominal tasks and contingency plans for failed actions. Although it is not the scope of this research, it would be useful if the planner could generate collision-free paths based on the world model.

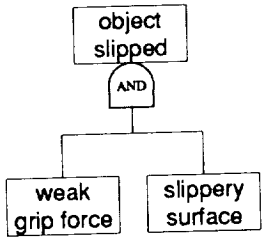
  One important feature that has been added to the planner is the capability to check commands that are initiated by the operator. Routine and tedious tasks often cause human fatigue and boredom.

This can lead to human error and a resultant hazardous situation. The user interface ensures that no erroneous or unsafe commands are given by the operator. And the planner will check whether the preconditions of the actions are satisfied, and the postconditions are acceptable.

- *User Interface*: The user interface plays an important role in the handling of failures in the system. As the system becomes more intelligent, it is expected that the demand from the operator will be reduced. Nonetheless, the user interface will always allow the operator to approve, query, and intervene a recovery plan. However, it is not enough for the operator merely to take control. The operator needs information such as the robot's status, position, and previous activities. At the time an operator must take control, he may not know such information. The system condenses this information and relay the important data to the operator control station.

- *World Model*: Information representing the work environment is integrated and assimilated in the world model. It is used by all components in the Task Planning System. The model is updated every time new information about the environment is received as a result of robot actions, sensory data, physical processes, and fault diagnosis.

- *Knowledge Base*: The knowledge base is the repository of all the rules and actions for planning, monitoring, diagnosis and recovery. One of the critical problems in knowledge base construction is the acquisition of expert knowledge. A fault tree analysis of the target robotic system will be performed and the resulting fault trees will be used as the basis for creating IF-THEN rules. For example, the subtree in Figure 5a can be translated into the rule in Figure 5b. The fault tree is helpful because it provides a visual representation of the way in which failures are propagated in the system.



(a)

```
IF weak grip force AND
   slippery surface THEN
   object slipped
```

(b)

Figure 5: A Fault Tree and its Translated Rule

CONCLUSIONS

A comprehensive strategy is described which integrates system level hardware/software fault tolerance with task level handling of uncertainties and unexpected events in robotic control. A prototype of the EDRB has been implemented using PC/AT-386 computers, ARCnet, and the QNX real-time operating system. Extensive evaluation has concluded that the resulting system tolerates a broad range of hardware, system software, and application faults, with a 200 millisecond guaranteed response time. The system is currently being rehosted to a VME-based multiprocessor system using 68040 single board computers and the VxWorks real-time operating system. It is expected that the faster hardware and system software will achieve the 5 milliseconds response time requried by the Manipulator Control System of the JPL Remote Surface Inspection System [15] to control a 7-DOF redundant manipulator arm.

The fault tolerant techniques developed in this research for building dependable robotic control systems can be used in applications which require a high degree of reliability and safety, such as servicing and inspection tasks in Space Station Freedom, maintenance and waste cleanup tasks in nuclear facilities, and patient monitoring and tending tasks in medical facilities.

ACKNOWLEDGEMENTS

The research described in this paper was partially carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. This work was funded by NASA Small Business Innovation Research (SBIR) Contract NAS7-1172.

REFERENCES

[1] M. H. Lee, D. P. Barnes, and N. W. Hardy, "Knowledge based error recovery in industrial robots," in *Proceedings of the International Joint Conference on Artificial Intelligence*, (Philadelphia, PA), pp. 824–826, 1983.

[2] M. Gini and R. Smith, "Monitoring robot actions for error detection and recovery," in *Proceedings of the NASA Conference on Space Telerobotics*, vol. III, (Pasadena, CA), pp. 67–78, Jan. 1987.

[3] D. E. Wilkins, "Recovering from execution errors in SPIE," in *Proceedings of the NASA Conference on Space Telerobotics*, vol. III, (Pasadena, CA), pp. 79–90, Jan. 1987.

[4] E. López-Mellado and R. Alami, "A failure recovery scheme for assembly workcells," in *Proceedings of 1988 IEEE International Conference on Robotics and Automation*, (Cincinnati, OH), pp. 702–707, May 1990.

[5] M. H. Lee, *Intelligent Robotics*. Open University Press, 1989.

[6] M. Hecht, J. Agron, H. Hecht, and K. H. Kim, "A distributed fault tolerant architecture for nuclear reactor and other critical process control applications," in *Digest of 21st International Symposium on Fault-Tolerant Computing*, (Montreal, Canada), pp. 3–9, June 1991.

[7] K. H. Kim and H. O. Welch, "Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications," *IEEE Trans. Computers*, vol. 38, pp. 626–636, May 1989.

[8] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Engineering*, vol. SE-1, pp. 220–232, June 1975.

[9] S. Hayati, T. S. Lee, K. S. Tso, P. G. Backes, and J. Lloyd, "A unified teleoperated-autonomous dual-arm robotic system," *IEEE Control Systems*, vol. 11, pp. 3–8, Feb. 1991.

[10] H. Seraji, "Configuration control of redundant manipulators: theory and implementation," *IEEE Trans. on Robotics and Automation*, vol. 5, pp. 472–490, Aug. 1989.

[11] C. A. Klein and C. H. Huang, "Review of pseudoinverse control for use with kinematically redundant manipulators," *IEEE Trans. on Systems, Man and Cybernetics*, vol. SMC-13, no. 3, pp. 245–250, 1983.

[12] H. Seraji and R. Colbaugh, "Improved configuration control for redundant robots," *Journal of Robotic Systems*, vol. 7, no. 6, pp. 897–928, 1990.

[13] I. J. Cox and N. H. Gehani, "Exception handing in robotics," *IEEE Computer*, pp. 43–49, Mar. 1989.

[14] P. G. Backes, K. S. Tso, S. Hayati, and T. S. Lee, "A modular telerobotic task execution system," in *Proceedings of 1990 IEEE International Conference on Systems Engineering*, (Pittsburgh, PA), pp. 511–514, Aug. 1990.

[15] S. Hayati, J. Balaram, H. Seraji, W. S. Kim, and K. Tso, "Remote surface inspection system," in *Proceedings of SOAR'92: The 6th Annual Space Operations, Applications, and Research Symposium*, (Houston, TX), Aug. 1992.