Proceedings of the 1997 IEEE
International Conference on Robotics and Automation
Albuquerque, New Mexico - April 1997

Providing Fault Tolerance for Active Vision Systems in Real-Time

Jeffrey A. Fayman and Ehud Rivlin
Computer Science Department
Technion—Israel Institute of Technology
Haifa 32000, Israel

Daniel Mossé
Computer Science Department
University of Pittsburgh
Pittsburgh, PA 15206

Abstract

The purpose of this paper is twofold: we first present a novel architecture for real-time active vision systems, and then enhance the architecture with a unified approach to fault tolerance. Our system is designed modularly in order to enable the flexible addition of hardware and software redundancy and also to allow reconfiguration when and where needed. This gives us the ability to handle faults in the context of Active Vision.

1 Introduction

Modern robotics applications are becoming more complex due to increasing numbers of resources to be controlled. An excellent example of this is the use of active vision in tasks which previously relied solely on static imaging. Although there are many advantages of active over passive vision such as improved robustness and elimination of ill-posed conditions [1, 2], it is known that the control of *active vision devices* (AVD's) is both complex and compute intensive as the AVD combines perception and action using multiple sensors and actuators.

In order to meet the computational demands of such systems and to support distributed sensors and actuators, multiple processors may be required. A distributed *real-time* architecture is needed to provide required processing while meeting application-specified timing constraints. Missing a deadline in such a system can lead to high-cost consequences, for example when autonomous robots handle radio-active materials or when human life is involved. Therefore, services offered must be available and provide guarantees despite the occurrence of faults (*fault tolerance*). In particular, for real-time systems, fault tolerance techniques must guarantee that timing constraints are met, in addition to functional correctness. For this reason, fault tolerance in real-time robotic environments is a hard problem.

Fault tolerance has been extensively studied for many general purpose applications. All of these techniques have a common element: *redundancy*. Several concepts developed for the support of general purpose fault tolerance can be adapted to execute in real-time environments. This is the case with broadcast and multicast communication primitives [3] and group memberships when managing replicated procedure calls [5, 4]. Recently, work in robotic fault-tolerance has appeared. At the system level, fault tolerance is researched in [13] where Tso et al. discuss a control system in which system level fault tolerance is integrated with task level handling of uncertainties and unexpected events. Visinsky et al. [14] present a layered fault tolerance framework which provides fault tolerance at the servo, interface and supervisor layers. However, to the best of our knowledge, real-time fault tolerance in active vision systems has not yet been addressed.

Many systems focus on a single type of redundancy, such as replicated hardware. However, a comprehensive framework is lacking to combine different types of redundancy according to user needs. In a previous work, we have developed a scheme that addresses task and application level fault tolerance [11]. At the task level, individual tasks have associated monitors that perform error detection and reporting, as well as some error handling. These monitors report to higher level monitors that maintain more semantic information and are able to broaden the fault coverage. At each monitoring level, there is an error handler (EH) associated with the monitor, that performs the error handling actions. At the application level, tasks are replicated and interconnected, creating a resilient application. The number of replicas created depends on the criticality of the application and of the individual tasks. Although there are no assumptions on the relative speed of the replicas, they may execute concurrently to minimize the error detection and recovery latencies.

In addition to task and application-level fault tolerance, a new approach for achieving fault tolerance at the system level, *scenario changes* [10], defines a mechanism by which the guarantees of applications accepted for execution may be revoked in a controlled fashion. It encompasses reconfiguration (i.e., re-distribution of applications in the resources of the system), *load-shedding*, which address a single form of recovery, timing faults or transient overloads, and *exception handling*.

In this paper, we present a distributed real-time architecture which provides the ability to combine the perceptive capabilities of active vision with the active capabilities of other robotic devices (Section 2) as well as several levels of fault tolerance (Section 3). We present a complex task in satellite retrieval and show how the architecture can provide solutions both for the tasks real-time requirements and fault tolerance (Section 4). In Section 5 our tests and results are shown. We close the paper with concluding remarks in section 6.

2 Basic Architecture

In this section we briefly review the basic architecture used in our work, which addresses both micro and macro levels in a unified manner. For a more thorough discussion, please see [6].

Our basic architecture provides both an interactive user interface/development component and a real-time component: the host and target machines, respectively, which can be seen in Figure 1. By the nature of the

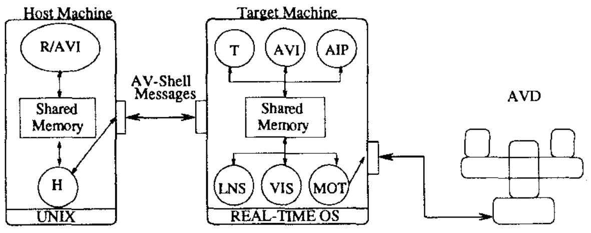


Figure 1: Basic System Architecture

different speed requirements necessary for interactive processing on the host machine and real-time processing on the target machines, these components are decoupled. The non-real-time interactive and developmental components of the architecture are separated from a low-level target module which uses hardware/software applicable to real-time activities. Communications between the two components is implemented as message passing using a well defined protocol [1].

AVI is implemented using the Robot Schemas (RS) notation introduced by Lyons [8]. RS provides a language for specifying process concurrency and captures the temporal and structural dependencies required to implement complex perception tasks. One of the novel features of our architecture is the inclusion of AVI on both the host and target machines. By distributing RS, we provide mechanisms for both rapid prototyping (when the AVI module on the host machine is used offline) and experimentation purposes (when the AVI modules on the target machines execute in real-time).

Composition of primitive routines into complex activities is achieved by traversing a parse tree built from an expression in *Robot Schemas* notation. We call a set of executing applications a *scenario*. The transition from one scenario to another can be implemented by the termination of one tree and the execution of another. A computing system executes exactly one scenario at a time. For example, an AVD consisting of a camera mounted on a robotic arm trying to intercept a moving target may consist of four major scenarios, namely *target location, tracking/interception, recovery* and *contingency* [2].

3 Task Scheduling and Fault Tolerant Architecture

Since several applications may be controlled by a single target system, in order to comply with the deadlines of tasks, we need to make sure that the target systems will not be overloaded with computations. To verify the feasibility of the schedules generated, we use a real-time scheduling algorithm, such as Rate Monotonic (RM) [7], Earliest Deadline First (EDF) [7], or time-line based algorithms [12].

The EDF or RM algorithms check whether a set of periodic tasks meets deadlines, according to the utilization of the task set. Consider a task set $T = \{t_1, t_2, \ldots, t_n\}$ with execution requirements $c_1, c_2, \ldots, c_n$ and periods $p_1, p_2, \ldots, p_n$. The utilization of task i is defined as the percentage of CPU needed for the task to execute, that is: $U_i = \frac{c_i}{p_i}$. A task set can be scheduled within their deadlines if the following relation holds:

$$U = \sum_{i=1}^{n} U_i \leq \begin{cases} n(2^{1/n} - 1) & \text{if } RM \\ 1 & \text{if } EDF \end{cases} \qquad (1)$$

In our implementation we use the RM method for the control of such applications, since there are few levels of priority needed in this system [3]. Each of the target systems implement RM to verify the feasibility of

[1]The interactive/development facilities of the architecture will not be discussed here. A thorough discussion of this component can be found in [6].

[2]Note that each of these scenarios may have applications distributed over several resources or sites.

[3]The main reason to use RM is that when many priority levels are needed, a hardware implementation of EDF is not possible.

the tasks submitted to it. Note, however, that the general approach is the same, regardless of the scheduling algorithm used.

In addition to guaranteeing timely execution, we provide redundancy at different levels of the system, in order to be able to tolerate different types of faults. This means that fault model is more flexible than if a single type of fault tolerance were being used. We allow, for example, one or more actuators to fail, one or more of the subcomponents of the target machines to fail, and so on.

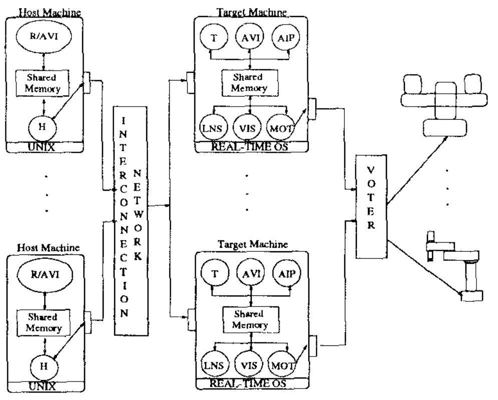


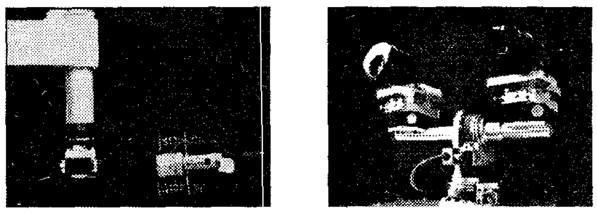
Figure 2: Logical System Architecture

The basic architecture described in Section 2 was used to obtain a fault-tolerant architecture, shown in Figure 2. This architecture provides fault tolerance at several levels. Using the language of [10], fault tolerance at the unit level is provided by replicating device hardware. For example, multiple vision boards as we will see in the example later in the paper. At the application level, fault tolerance is provided by replicating the hardware and software of the target machine. Here, faults related to processing unit failure including CPU, vision card, motor drivers, etc, can be tolerated. At the system level, fault tolerance is achieved by replicating the hardware and software components so that failure of a host machine does not prohibit continued system execution [4].

The AVI's in the target machines parse and execute the trees for AVI implementation. After an output is generated, the target machines exchange information about the generated output in order to verify that all replicated target machines for a device computed the same values. This is done by the *voter*, which produces a unique message from those that it receives and sends a single message to the AVD/actuator. The voter also determines if a fault has occurred in one of the target modules. It is the responsibility of the voter to determine which (if any) of the target modules has failed and to perform appropriate system reconfiguration in such an event. Failure criteria include timeouts and differing input from duplicate target modules.

The voter can be located in several physical locations, namely in the actuator, in a separate machine, or implemented in a distributed fashion among all the target systems. The different options present different overheads, advantages and disadvantages. For example, the centralized voter residing in one of the target machines becomes a bottleneck for the voting procedure and represents a single point of failure (if that particular target machine fails, the entire scheme fails, since there is no output message to the AVD/actuator). To solve these problems, one can locate the voter inside the AVD/actuator, but this requires either modifying the hardware or modifying the software at the AVD/actuator, neither of which is simple to achieve.

4 Experimental Task



(a) (b)

Figure 3: (a) Satellite and Docking Vehicle; (b) Active Vision Head

Our experimental configuration consists of an AdeptOne robot arm (depicted as the docking vehicle on the left part of Figure 3(a)) with a camera mounted on it, and a satellite being held by a Scorbot-ER9 robot arm (on the right part of Figure 3(a)). Additionally, there is an *active vision head* with 4 degrees of freedom (shown in Figure 3(b)).

The docking vehicle attempts to intercept the moving satellite by adjusting its position according to satellite motion information provided by the camera mounted on the arm. Additionally, the active vision head provides information to help guide the docking process by evaluating the satellite's trajectory. As a result of the satellite's motion and due to the nature of the control application, deadlines are imposed on each of the tasks that must be carried out.

Let us assume that in the operation of such a docking procedure there are fourteen scenarios, namely *target location, long-range tracking/docking, close-range*

[4]Details of the host machine hardware and software can be found in [6].

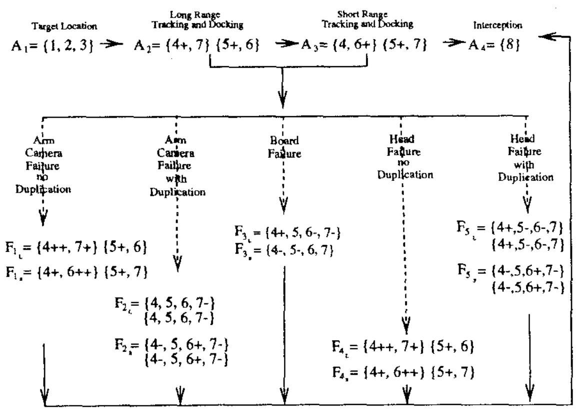

Target Location
Long Range Tracking and Docking
Short Range Tracking and Docking
Interception
A1= {1, 2, 3}
A2= {4+, 7} {5+, 6}
A3= {4, 6+} {5+, 7}
A4= {8}
Arm Camera Failure no Duplication
Arm Camera Failure with Duplication
Board Failure
Head Failure no Duplication
Head Failure with Duplication
F1L= {4++, 7+} {5+, 6}
F1S= {4+, 6++} {5+, 7}
F2L= {4, 5, 6, 7-} {4, 5, 6, 7-}
F2S= {4-, 5, 6+, 7-} {4-, 5, 6+, 7-}
F3L= {4+, 5, 6-, 7-}
F3S= {4-, 5-, 6, 7}
F4L= {4++, 7+} {5+, 6}
F4S= {4+, 6++} {5+, 7}
F5L= {4+,5-,6-,7} {4+,5-,6-,7}
F5S= {4-,5,6+,7-} {4-,5,6+,7-}

Figure 4: Scenario applications and transitions. The recovery/contingency scenarios are labeled *L* if transition is from scenario *A2* and *S* if transition is from scenario *A3*.

tracking/docking, interception and several scenarios for *recovery/contingency.*

appl	functionality	period	time	CPU
1	motion detect	100	25.0	25.0%
2	recog./verif.	–	100.0	–
3	saccade	–	10.0	–
4-	smooth pursuit	110	13.3	12.1%
4	smooth pursuit	110	19.4	17.6%
4+	smooth pursuit	110	30.1	27.4%
4++	smooth pursuit	110	47.2	43.0%
5-	trajectory est.	750	100.0	13.3%
5	trajectory est.	750	162.0	21.6%
5+	trajectory est.	750	322.0	43.0%
6-	docking	110	13.3	12.1%
6	docking	110	19.4	17.6%
6+	docking	110	30.1	27.4%
6++	docking	110	47.2	43.0%
7-	TTC	100	5.0	5.0%
7	TTC	100	20.0	20.0%
7+	TTC	100	25.0	25.0%
8	interception	–	10.0	10.0%

Table 1: Applications and timing constraints for AVD-aided docking

Let A_i be a scenario composed of a set of applications (denoted by numbers, described in Table 1) and "$\Rightarrow$" define the transitions between scenarios. The symbols "+" and "–" following the applications indicate varying times required to perform the same activity with varying quality: more and less accuracy, with more or less CPU required, respectively. Typically, the more important tasks will be carried out by "+" applications and the less important by "–" applications. The symbol "++" indicates extra time with diminished accuracy, due to the hardware incompatibility. Figure 4 shows the sets of applications in each scenario as well as the transitions from one scenario to the next. Since these applications are extremely compute-intensive, they run in parallel in a dual-CPU system (thus the two sets of applications).

In the *target location* scenario (A_1), the head is executing a motion detection procedure. When a moving target is detected, the verification procedure is run and a saccade is invoked; therefore, this scenario consists of applications $\{1,2,3\}$. A scenario change is triggered when the satellite is fixated, moving to the *long range tracking/docking* scenario (A_2). This is an example of changes that include total reconfiguration of the task sets being executed: $\{1,2,3\} \Rightarrow \{4+,7\}\{5+,6\}$. In this scenario and the next (*short range tracking/docking* (A_3)), four tasks are being executed: smooth pursuit, docking, trajectory estimation and time-to-contact. Smooth pursuit is used by the head to track the satellite and improve satellite trajectory estimation. The camera mounted on the AdeptOne computes the needed information to control the arm for the docking procedure while time-to-contact measures the time until contact with the satellite is made.

Long range tracking/docking is in effect from the time the satellite is recognized and fixated upon until the time-to-contact module determines that the distance from the docking vehicle to the satellite is less than some threshold (70cm in our experiments) at which point a scenario change is triggered to *short range tracking/docking* (A_3). While the distance between the docking vehicle and the satellite is greater than the threshold, trajectory estimation plays a greater role in the control of the system as we are positioning the docking vehicle relative to the satellite. Therefore, we use high-quality algorithms (in terms of accuracy) for smooth pursuit and trajectory estimation. Once the distance to the satellite drops below the threshold, the results of the docking and time-to-contact modules become more important than trajectory estimation as the trajectory should not change too much but we need high accuracy in the time-to-contact computations because the distance from the satellite is very small and errors can be catastrophic (i.e., overshooting and hitting the satellite). For this reason, tracking/docking has been divided into two scenarios, long range and short range.

The last normal scenario change occurs in the case the time-to-contact approaches 0: the system changes to the *interception* scenario (A_4). However, we must also consider that several faults can occur in the system: each of the boards in the target systems can fail or each of the cameras can fail. Therefore, in what follows we examine the different scenarios that are reached when each of these faults occur.

If a vision board in one of the target machines fails, the system must adapt to the new situation and may need to gracefully degrade the quality of its services. This can be implemented with scenario changes: $\{4+,7\}\{5+,6\} \Rightarrow \{4+,5,6-,7-\}$ (i.e., $A_2 \Rightarrow F_{3_L}$) in the case of board failure while in the *long range tracking/docking* scenario, and $\{4,6+\}\{5+,7\} \Rightarrow \{4-,5-,6,7\}$ (i.e., $A_3 \Rightarrow F_{3_S}$) when board failure occurs while in the *short range tracking/docking* scenario. Note that in this case the graceful degradation also includes reconfiguration, since all applications will run on a single vision board. In this situation, since there is a single non-failed board, the procedures are less accurate but will be able to complete their required tasks (probably in a longer time interval or with more tries). If the failed component is replaced and/or fixed, the scenarios again change from recovery to tracking/interception, that is, $\{4+,5,6-,7-\} \Rightarrow \{4+,7\}\{5+,6\}$ (i.e., $F_{3_L} \Rightarrow A_2$), or $\{4-,5-,6,7\} \Rightarrow \{4,6+\}\{5+,7\}$ (i.e., $F_{3_S} \Rightarrow A_3$) which represents the re-allocation of tasks.

If a camera fails, there are several possible cases. Due to lack of space, we will only discuss the case when the arm camera fails and application degradation and duplication are performed. In this case, since the docking procedure needs to be carried out with guidance from the head, more computationally intensive procedures must be executed (with only information from head cameras). If the arm camera fails while in *long range tracking and docking*, we use the applications $4++$ and 7+, instead of 4+ and 7. In the case of arm camera failure while in *short range tracking and docking*, we would like to use applications 5+ and $6++$. However, these applications have a utilization of 86.0% which exceeds the limit for the Rate Monotonic scheduling algorithm. Therefore, we must reconfigure the application allocation. In this case, we would use scenario $\{4+,6++\}\{5+,7\}$.

5 Experiments

The goal of our experiments is to verify the effectiveness of our architecture. To show this, we have implemented the experimental configuration described in section 4. In what follows, we describe how we have implemented on our architecture four techniques for performing smooth pursuit and docking as well as two techniques for computing time-to-contact.

5.1 Motion analysis

We have implemented and tested four motion analysis techniques for smooth pursuit and docking. Each technique possess differing characteristics of both time and quality. To identify these characteristics, we measured the ability of each technique to track various combinations of objects and backgrounds. In all, we tested six such combinations where objects range from a simple blob to the satellite, and backgrounds range in complexity from a constant black background (as found in space) to backgrounds with randomly placed objects. Tracking module timing and module success ratio are given in the table of Figure 5.

Module	Applic.	Time	Succ. Ratio
Blob Tracking	4−, 6−	13.3	**37%**
Edge Tracking	4, 6	19.4	**40%**
Template Mat.	4+, 6+	30.1	**43%**
Image Diff.	4++, 6++	47.2	**47%**

Figure 5: (a) Module Timing and Success Ratio. Speed at which each module runs independently. Success ratio is calculated as the percentage of successful runs over the entire set of experiments.

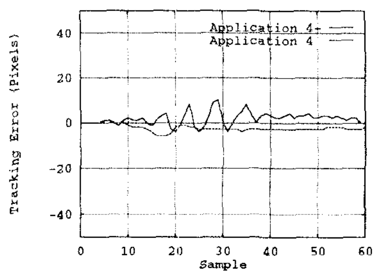


Figure 6: Tracking Results from Two Motion Modules

Figure 6 shows the tracking results of applications 4− and 4. 4− is a faster but less accurate algorithm. It is clear from the figure that the faster algorithm leads to a degradation in tracking stability.

5.2 Time-To-Contact

We have implemented two time-to-contact (TTC) modules. One module is divergence based while the other is model based. It was shown in [9] that the rate of change of the area of an object, divided by the area, is proportional to the divergence. Using the divergence we can approximate the TTC. In the model based approach, we track known features to find the changes in the apparent length. These changes were used to approximate TTC in a similar manner.

Figure 7(a) shows the performance of the model based technique (application 7−) over the entire satellite motion while figure 7(b) shows the performance of the divergence based technique (application 7) over the same motion. It is clear from the figures that the divergence based technique leads to more stable estimates of TTC than the model based technique. TTC stability becomes increasingly important the closer the docking station is to the satellite. We would like to avoid a situation such as the TTC module giving a false measurement of a distance in short range. This can lead to a failure in docking as well as damage both to the satellite and docking vehicle. In order to avoid this situation, we switch scenarios $A_2 \Rightarrow A_3$ when the satellite is at a distance of 70cm.

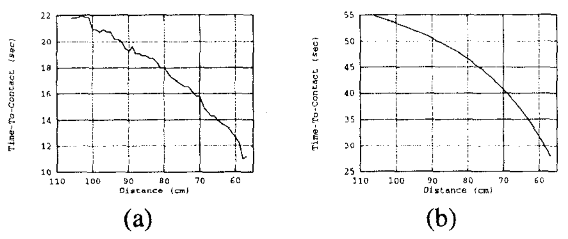


Figure 7: (a) Model Based TTC (application 7−) at 5cm/sec; (b) Divergence Based TTC (application 7) at 2cm/sec

5.3 Scenario switching

When a fault is detected, the system must appropriately reconfigure tasks according to the scenario changes. Task reconfiguration introduces overhead in the system which must be accounted for. In order to quantify task reconfiguration overhead, we have experimented with various legal scenario changes according to our experimental scenario. The results of the timing analysis are presented in Table 2. We performed each scenario switch 1000 times accumulating the time and then averaging the results to obtain the values in Table 2.

Scenario	A_2	A_3	$F3_L$	$F3_S$
A_2	.16	.16	.16	–
A_3	.16	.16	–	.002
$F3_L$	.159	.16	.16	.002
$F3_S$	.16	.16	.16	.002

Table 2: Timing Analysis for Scenario Switching

6 Summary

We presented two main elements in this paper: a novel architecture for real-time active vision systems, and enhancement of the architecture with a unified approach for fault tolerance. The integration of real-time fault-tolerant active vision systems into robotics applications both enhances the functionality of such systems as well as ensuring their ability to perform well despite the occurrence of faults.

We designed our architecture with inherent fault-tolerant properties in order to cope with real-time and robotics requirements. Such design was used to enable flexible fault tolerance, in terms of software and hardware redundancy and reconfiguration. With our fault tolerance approach, the software functions *and* hardware devices are protected against failures, and allow tolerance to faults at different levels of abstraction.

References

[1] Y. Aloimonos, I. Weiss, and A. Bandyopadhyay. Active vision. In *International Journal on Computer Vision*, pages 333–356, 1987.

[2] R. Bajcsy. Active perception vs passive perception. In *Proceedings of the Third IEEE Workshop on Computer Vision*, pages 55–59, Bellaire, Michigan, 1985.

[3] K. Birman. Replication and Fault-Tolerance in the ISIS System. In *10th ACM Symp on Oper Syst Principles*, pages 63–78, WA, Dec 1985. ACM.

[4] A. Borg, J. Baumbach, and S. Glazer. A Message Passing System Supporting Fault Tolerance. In *9th AMC Symp on Oper Syst Principles*, pages 90–99, NH, Oct 1983. ACM.

[5] Eric Cooper. Replicated Distributed Programs. In ACM, editor, *10th ACM Symp on Oper Syst Principles*, pages 63–78. ACM, Dec 1985.

[6] J. Fayman, E. Rivlin, and H. Christensen. The active vision shell. Technical Report CIS 9510, Technion - Israel Institute of Technology, April 1995.

[7] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment. *jacm*, pages 46–61, January 1973.

[8] D.M. Lyons and M.A. Arbib. A task-level model of distributed computation for sensory-based control of complex robot systems. In *IFAC Symposium, Robotic Control*, Barcelona, Spain, November 1985.

[9] S. Maybank. Apparent area of a rigid moving body. *Image and Vision Computing*, 5(2):111–113, May 1987.

[10] D Mossé. Mechanisms for System-Level Fault Tolerance in Real-Time Systems. In *Int'l Conf on Robotics, Vision, and Parallel Processing for Industrial Automation*, June 1994.

[11] D. Mossé. Creating Resilient Real-Time Applications. In *20th IFAC/IFIP Workshop on Real-Time Programming*, Nov 1995.

[12] J. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Systems. *IEEE Software*, May, 1991.

[13] K.S. Tso, M. Hecht, and N.I. Marzwell. Fault-tolerant robotic systems for critical applications. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Atlanta, Georgia, May 1993.

[14] M.L. Visinsky, J.R. Cavellaro, and I.D. Walker. A dynamic fault tolerance framework for remote robots. *IEEE Transactions on Robotics and Automation*, 11(4):477–490, August 1995.