Real-Time Active Vision with Fault Tolerance

Jeffrey A. Fayman and Ehud Rivlin
Computer Science Departament
Technion, Israel Institute of Technology
Haifa 32000, Israel
{jefff,ehudr}@cs.technion.ac.il

Daniel Mossé
Computer Science Departament
University of Pittsburgh
Pittsburgh, PA 15206
mosse@cs.pitt.edu

Abstract

The active vision paradigm couples perception and action at several different levels. The effective use of active vision in complex robotic tasks requires that these levels operate both independently and cooperatively, reliably and in real-time. In this paper, we present a system for real-time active vision with fault tolerance. The system provides a vocabulary of active vision routines along with the means for composing the routines into continuously running perception-action processes. A novel architecture which enables the integration of the perceptive capabilities of real-time active vision with the active capabilities of other robotic devices is presented. We then enhance the architecture with a unified approach to fault tolerance and present results from experiments and simulations.

1 Introduction

It is well known in active vision that the ability to move the visual sensory system leads to improved robustness and the elimination of ill-posed conditions in several computer vision problems. However, the interface to *active vision devices* (AVD's) as well as their reliable control are both complex and compute intensive as the AVD's combine perception and action using multiple sensors and actuators.

In order to effectively exploit the benefits of active vision in complex robotic systems, the capabilities of AVD's must be directly accessible to the user in a convenient manner. Moreover, a distributed *real-time* architecture is needed to deliver required processing while meeting application-specified timing constraints. Missing a deadline in a 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.

In this paper we present a system which provides a powerful interface to AVD's (section 3), and a distributed real-time architecture for perception and action tasks with several levels of fault-tolerance (section 4). We present a complex task which uses active vision for object capture and show how the system can provide solutions both for the tasks real-time requirements and fault tolerance (section 5). In section 6 our tests and results are shown.

2 Related Work

Although many researchers have looked into problems of combining perception and action at the micro level of the AVD, relatively little work has been done in integrating active vision into more complex systems. In [6], Crowley and Christensen discuss a sophisticated architecture and system called "VAP/SAVA" for the integration and control of a real-time active vision system. Their architecture was designed to accommodate a continuously operating process, but does not provide tolerance to faults. It is mainly a hardware approach to real-time imaging.

Fault tolerance has been extensively studied for many general purpose applications. All of these techniques have a common element: *redundancy*. Several concepts devel-

1015-4651/96 $5.00 © 1996 IEEE
Proceedings of ICPR '96

oped 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], group memberships when managing replicated procedure calls [5, 4], as well as error detection in both control flow errors [15] and functional errors [13].

Fault tolerance research in real-time environments has been mostly been addressed in a static way [2], by using hardware replicas [8, 14] or software alternatives [1]. A hybrid system [9, 18] combines hardware redundancy with the ability to distribute computations, but still lacks in flexibility. Fault-tolerant techniques are essential for successful missions [16].

3 An Active Vision Interface - The AV-Shell

In order to fully exploit the benefits of active vision in complex robotic tasks, a convenient interface to AVD's and the other robotic devices is required. The *Active Vision Shell* (AV-Shell), is a system which provides this interface through a vocabulary of active vision routines and the means for their composition. The AV-Shell [1] is an interactive program written in C under the UNIX operating system. It interprets and executes active vision and robotics related commands called "AV-Shell commands" in much the same way that a shell such as c-shell interprets and executes system related commands. Included in the set of commands is an extensive set of algebraic operations including matrix and vector manipulation and various arithmetic operations and standard mathematic functions for scalars all of which are used extensively in robotics work. Also included in the AV-Shell are a set of active vision routines which are made up of three "levels"; (1) high-level activity routines supporting activities such as *fixation*, *smooth pursuit*, and *stabilization*; (2) medium-level basic process routines such as *dynamic accommodation*, *aperture control*, *saccade*, *vergence and disparity estimation* and *peripheral and foveal motion detection*; and (3) low-level primitive routines such as *histogram* and *correlation*. In this configuration, processes at higher levels are made up of processes at lower levels. For composing lower level commands into higher level activities, the AV-Shell adopts the Robot Schemas *RS* process composition model [11] which provides a language for specifying process concurrency and captures the temporal and structural dependencies required to implement complex perception tasks. The following are examples of two active vision routines in the notation of the AV-Shell: (a thorough discussion of the AV-Shell can be found in [7]).

accommodate

Accommodate provides a quality measure as to the quality of focus obtained by a camera. It can be used in synchronous recurrent composition with the **camera -f** command to implement continuous focus control.

motion [*-p*][*-f size*]

The motion command implements both peripheral and foveal motion detection. If the *-p* option is used, motion causes a continuous monitoring for peripheral motion. If *-f* is used, motion causes a continuous foveal motion detection in a fovea of $size \cdot size$.

4 Architecture

The AV-Shell provides a consistent interface to a wide range of active vision and robotic devices. In addition to the interface, complex perception-action robotic applications, which integrate active vision with other robotic devices, demand sophisticated architectures. We must view perception and action on a micro-level looking at the AVD as it operates in an independent perception-action cycle to control its own movements and at a macro-level where the information provided by the AVD is used to drive the actions of the other robotic devices.

By the nature of the different speed requirements necessary for AV-Shell interactive processing and device level real-time processing these components are decoupled. The non real-time interactive and developmental components of the architecture are implemented in the "host machine" while the real-time components are implemented in the low-level "target machine" which uses hardware/software applicable to real-time activities. This logical configuration is illustrated in figure 1. Communications between the two components is implemented as message passing using a well defined protocol. On the target machine, the target server **T** receives messages from the host server **H** and uses the messages to update shared memory on the target machine which is then interpreted by other target processes. The Active Image Processing process **AIP** is responsible for data filtering and fusion. The AV-integration process **AVI** is re-

[1] The AV-Shell is an extension of a system called the R-Shell [17] to include AVD's.

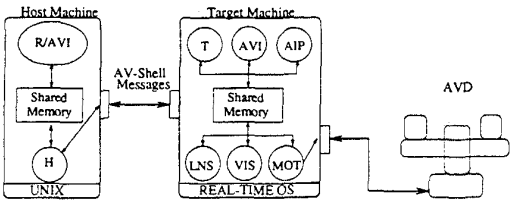

Figure 1. Basic System Architecture

sponsible for composing the routines mentioned in section 3 into continuously running perception-action processes. The motion control process **MOT** provides servo control to the actuators. Vision processes **VIS** carry out the pre-attentive image processing, while the **LNS** process provides lens control. The AVI is implemented using the *RS* model mentioned in section 3. One of the novel features of this 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).

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 are not overloaded with computations. To verify the feasibility of the schedules generated, we use the Earliest Deadline First (EDF) [10] algorithm although our approach is the same regardless of the scheduling algorithm.

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 the 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. The basic architecture described in figure 1 was augmented to obtain the fault-tolerant architecture, shown in Figure 2. This architecture provides fault tolerance at several levels. Using the language of [12], fault tolerance at the *unit level* is provided by replicating device hardware. This includes, 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 hardware and software components so that failure of a host machine does

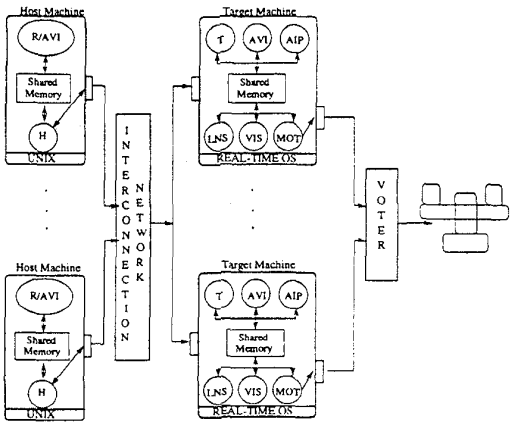

Figure 2. Augmented System Architecture

not prohibit continued system execution.

Commands arriving at the target machines are parsed and executed and an output generated. Thereafter, 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 occured in one of the target modules. It is the task 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 timeout and differing input from duplicate target modules.

5 Experimental Scenario

Our experimental set-up, which is depicted in figure 3 consists of an *active vision head* with 4 degrees of freedom, an AdeptOne robot arm with a camera mounted on it, and a toy train moving on a track.

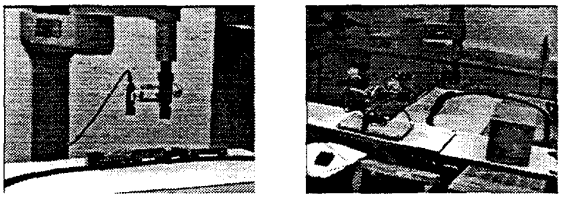

Figure 3. Experimental Scenario

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

We call a set of executing applications a *scenario*. Let us assume that in the operation of our grasping task there are eight scenarios, namely *target location, tracking/interception, grasping* and several *recovery/contingency*. 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). The applications control different services, described in the table of figure 5.

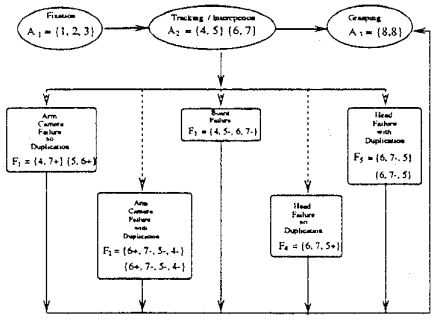

Figure 4. Applications and Transitions

appl	functionality	period	exec time	% of CPU needed
1	motion detection	-	25	-
2	recognition/verification	-	100	-
3	saccade	-	10	-
4-	smooth pursuit	150	30	20%
4	smooth pursuit	100	30	30%
5-	trajectory estimation	750	100	13%
5	trajectory estimation	750	300	40%
5+	trajectory estimation	750	400	53%
6-	interception	250	30	12%
6	interception	250	100	20%
6+	interception	250	125	50%
7-	time to contact	500	40	8%
7	time to contact	500	250	50%
7+	time to contact	500	300	60%
8	grasping	-	10	-

Figure 5. Applications and Timing Constraints

The symbols "+" and "–" following the applications indicate varying times required to perform the same activity with varying quality due to the hardware incompatibility and diminished accuracy, respectively.

Let $\{A_1, A_2, \ldots, A_k\}$ represent a scenario and "$\Rightarrow$" define the transitions between scenarios. In the *fixation* scenario, the head is executing a motion detection procedure. When a moving target is detected, the verification procedure is run and a saccade is executed; therefore, this scenario consists of $\{1, 2, 3\}$. A scenario change is triggered when the target is positively identified, moving to the tracking/interception scenario. This is an example of changes that include total reconfiguration of the task sets being executed: $\{1, 2, 3\} \Rightarrow \{4, 5\}\{6, 7\}$. In this scenario, smooth pursuit is running, and the head tries to estimate the trajectory of the train. The camera mounted on the AdeptOne computes the needed information to control the arm for interception. Another important quantity that is computed from the images that are sent by the arm-mounted camera is the *time to contact*, which influences the control of the arm. In the case that the time to contact approaches 0, the system changes to the *grasping* scenario ($\{8\}\{8\}$). However, we must also consider that several faults can occur in the system: each of the cameras can fail, each of the boards in the target systems can fail, or the components in the host can fail. In what follows, we look at the case of camera failure.

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. The other cases in figure 4 are analogous. In this case, the user wants the extra ability of tolerating more faults, such as the failure of one of the vision boards. Therefore, a degraded mode is entered, with scenario $\{4-, 5-, 6+, 7-\}\{4-, 5-, 6+, 7-\}$. That is, smooth pursuit and trajectory estimation are done using less accurate algorithms or lower sampling rates. The same holds for the time-to-contact procedures: the head camera tries to guide the interception procedure based on a sideways view, using the $6+$ algorithm. The scenario is duplicated in both vision boards for fault tolerance.

6 Experiments

We have verified the effectiveness of our architecture by measuring the real-time and fault-tolerant properties of the system.

In particular, we checked the accuracy when the system transitions from scenario A_2 to scenario F_2. We tested this

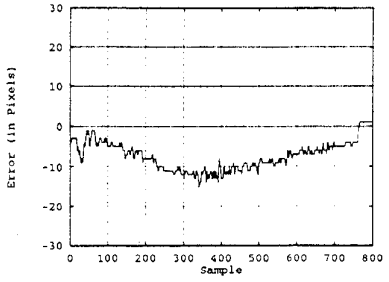

Figure 6. Normal Tracking of Scenario A_2

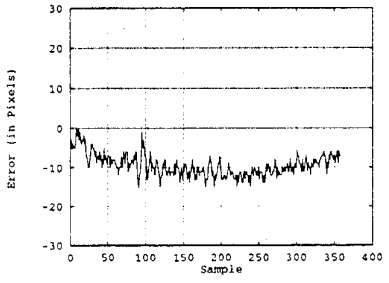

Figure 7. Degraded Tracking of Scenario F_2.

scenario for graceful degradation in terms of the accuracy of the smooth pursuit procedures. Since the arm camera failed, the two boards will carry out the same (but degraded) computations. The results of these experiments is given in figures 6 and 7. In scenario A_2 (normal smooth pursuit), a sampling rate of 100 milliseconds was used. In scenario F_2 (degraded smooth pursuit), a lower sampling rate was used according to the specification of degraded mode (see Figure 5). This reflects the fact that the vision card, carrying out the reconfigured scenario, is devoting less time to pursuit. It is clear from the figures that the degraded algorithm leads to a degradation in tracking stability.

To check the feasibility of scheduling the different scenarios on our system hardware, we measured the latency of scenario switching. 1000 scenario switches were executed and the latency measured and averaged. This yielded an average latency of 89 milliseconds.

In this paper, we presented two main elements: an interface to active vision devices, and a novel architecture for real-time active vision systems with 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 insuring their ability to perform well despite the occurrence of faults.

References

[1] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Trans Software Engineering*, SE-11(12):1491–1501, Dec 1985.

[2] S. Bennett and G. S. Virk (eds). *Computer control of real-time processes*. Institution of Electrical Engineers, London, UK, 1990.

[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.L. Crowley and H.I. Christensen. *Vision as Process*. Springer-Verlag, January 1995.

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

[8] K. Kant. Software Fault Tolerance in Real Time Systems. *Information Sciences*, 42(3):255–282, Aug 1987.

[9] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwalbl, Ch. Senft, and R. Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The MARS Approach. *IEEE Micro*, February 1989.

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

[11] 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.

[12] 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.

[13] S. K. Oh and G. MacEwen. Toward Fault-tolerant Adaptive Real-Time Distributed Systems. External Technical Report 92-325, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, January 1992.

[14] F. J. Pollack and K. C. Kahn. The BIIN Mission Critical Computer Architecture. In *Proc. of Workshop on Operating Systems for Mission Critical Computing*, Sept. 1989.

[15] M. A. Schuette and J. P. Shen. Processor Control Flow Monitoring using Signatured Instruction Streams. *Transactions on Computers*, C-36(3):264–276, March 1987.

[16] Reid Simmons. Concurrent planning and execution for a walking robot. Technical report, CMU, 1990.

[17] M.I. Vuskovic, A.L. Riedel, and C.Q. Do. The robot shell. *International Journal of Robotics and Automation*, 3(3):165–175, 1988.

[18] J.H. Wensley *et al.* The Design, Analysis, and Verification of the SIFT Fault Tolerant System. In *2nd Internatinal Conference on Software Engineering*, pages 458–469. IEEE, Computer Society, 1976.