Real-Time Imaging **4**, 143–157 (1998)

FT-AVS: a Fault-tolerant Architecture for Real-Time Active Vision

The nature of most problems addressed by robotics requires that robotic systems possess real-time properties. Additionally, as a result of steady increases in power and decreases in the cost of technology, it has become feasible to integrate sophisticated vision systems into robotic tasks. This can be seen by the recent interest in active vision.

The purpose of this paper is two-fold: 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. Further, the distribution of software on the available hardware is such that users can utilize a dual-mode of execution (simulation and rapid prototyping). Lastly, the tests we ran on the implemented architecture in order to validate the results of the experimentation and simulations show a good correlation of parameters.

Jeffrey A. Fayman[*], Ehud Rivlin[*] and Daniel Mossé[†]

[*]*Computer Science Department, Technion, Israel Institute of Technology, Haifa 32000, Israel*
E-mail: {jefff,ehudr}@cs.technion.ac.il
[†]*Computer Science Department, University of Pittsburgh, Pittsburgh, PA 15206*
E-mail: mosse@cs.pitt.edu

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 [1,2], it is known that the control of *active vision devices* (AVD's) is both complex and computationally 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 radioactive 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 difficult problem to overcome.

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 as well as several levels of fault tolerance. We present a complex task in satellite retrieval and show how the architecture can provide solutions both for the task's real-time requirements and fault tolerance. Before describing our new approach, we briefly review, in the next section, representative work related to this paper.

Related Work

In recent years, active vision has become an area of intensive research. Active vision, which was first introduced in [2] and later explored in [1] and [3], requires explicit control of the sensory system to improve robustness and eliminate ill-posed conditions. It has been shown that by actively controlling the vision system, many classic computer vision problems become easier to solve [1].

Although many researchers have looked into problems of combining perception and action at the micro level of the active vision device, relatively little work has been done in integrating active vision into more complex systems. In [4], 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 as 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 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 [5] and group memberships when managing replicated procedure calls [6,7] as well as error detection in both control flow errors [8] and functional errors [9].

Fault-tolerance research in real-time environments has mostly been addressed in a static way [10], by using hardware replicas [11,12] or software alternatives [13]. A hybrid system [14,15] combines hardware redundancy with the ability to distribute computations, but still lacks in flexibility. Fault-tolerant techniques are essential for successful missions [16].

Some recent work dealing with fault tolerance in robotics has been carried out by several researchers. At the system level, fault tolerance is researched in [17], 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*. [18] present a layered fault-tolerance framework which provides fault tolerance at the servo, interface and supervisor layers. Their work makes use of analytical redundancy, where measured system outputs are compared with the corresponding expected outputs derived from a model of the system behavior. Our work differs from the above in that we present a practical architecture which integrates the aspects of real-time, multi-level fault tolerance, and the ability to control multiple active vision and robotic devices. To the best of our knowledge, real-time fault tolerance in active vision systems has not yet been addressed.

Model of computation

Many systems focus on a single type of redundancy, such as replicated hardware. However, a comprehensive framework to combine different types of redundancy according to user needs is lacking. In previous work we have developed a scheme that addresses task- and application-level fault tolerance [19]. 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 sematic information and are able to broaden the fault coverage. At each monitoring level there is an error handler (EH) associated with the monitor which 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.

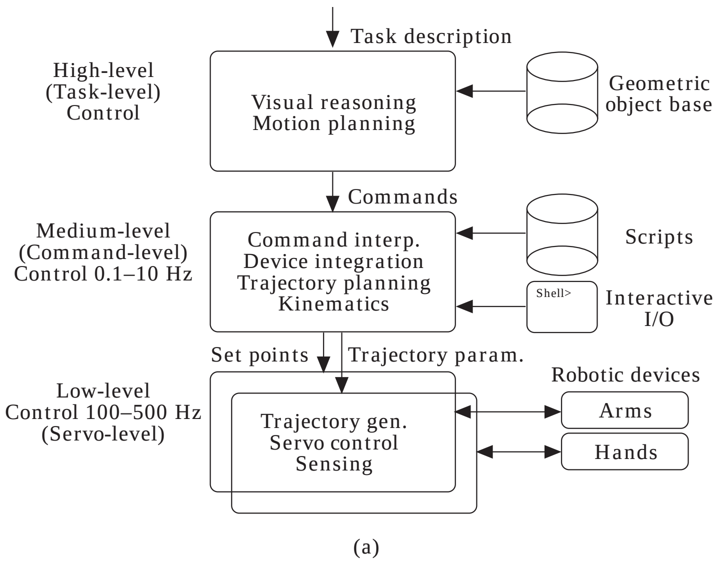
In addition to task- and application-level fault tolerance, a new approach for achieving fault tolerance at the system level, *scenario changes* [20], defines a mechanism by which the guarantees of applications accepted for execution may be revoked in a controlled fashion. It encompasses reconfiguration (i.e. redistribution 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*, such as in Stoyenko's seminal work [21]. The implementation of such scenario-switching techniques is done transparently to the user.

Architecture

Complex perception-action robotic applications, which integrate the perceptive capabilities of active vision with the active capabilities of a variety of robotic devices, demand sophisticated architectures. We must view perception and action on a micro level, looking only at the AVD as it must operate in an independent perception-action cycle to control its own movements. At the macro level, the information provided by the AVD is used to drive the actions of the other robotic devices. In this section we introduce the general architecture used in our work, which addresses both micro and macro levels in a unified manner. In Section 4 we discuss how fault tolerance is added to the general architecture, using the scenario changes mechanism [20].

The control of robotic devices is typically divided into three levels: **high**, **medium** and **low**. Figure 1(a) depicts the relationship among these three levels.

While this hierarchy is appropriate for devices such as manipulators, it is insufficient for the active image processing and composition of active vision routines required by active vision. In order to modularize these various levels of processing, we propose that, in the case of AVD's, two


Task description
High-level (Task-level) Control
Visual reasoning Motion planning
Geometric object base
Commands
Medium-level (Command-level) Control 0.1–10 Hz
Command interp. Device integration Trajectory planning Kinematics
Scripts
Shell>
Interactive I/O
Set points
Trajectory param.
Low-level Control 100–500 Hz (Servo-level)
Trajectory gen. Servo control Sensing
Robotic devices
Arms
Hands
(a)

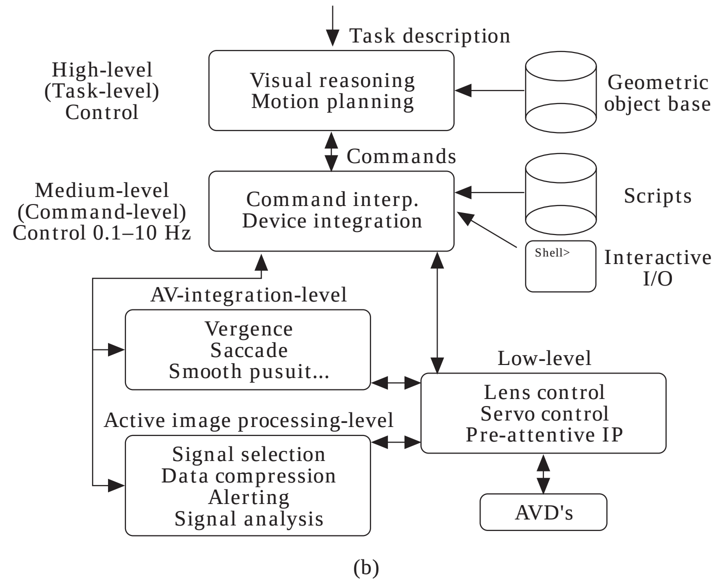

Task description
High-level (Task-level) Control
Visual reasoning Motion planning
Geometric object base
Commands
Medium-level (Command-level) Control 0.1–10 Hz
Command interp. Device integration
Scripts
Shell>
Interactive I/O
AV-integration-level
Vergence Saccade Smooth pusuit...
Low-level
Lens control Servo control Pre-attentive IP
Active image processing-level
Signal selection Data compression Alerting Signal analysis
AVD's
(b)

Figure 1. (a) Typical control hierarchy; (b) augmented control hierarchy.

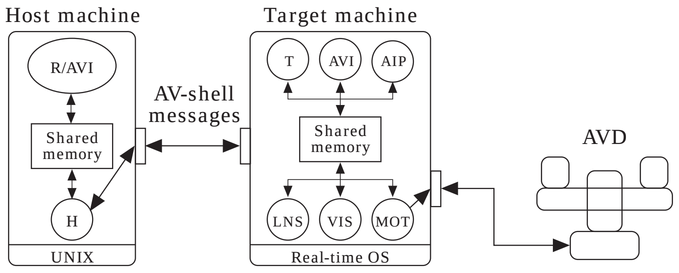


Figure 2. Basic system architecture.

additional levels are added to the control hierarchy of Figure 1(a). The new levels are called the *Active Vision Integration Level* (AVIL) and the *Active Image Processing Level* (AIPL). The general hierarchy, augmented with the new levels, is shown in Figure 1(b).

As shown in Figure 1(b), the low level is responsible for providing pre-attentive image processing. The AIPL performs filtering and fusion to provide data to the set of attentive routines found in the AVIL. The AVIL in turn is responsible for composing the higher level attentive routines, according to changing requirements, and also for compensating system delays. In our implementation, all three of these levels are accessible from the command interface at the medium level, facilitating development and simulation.

The basic architecture used in this work provides a logical instantiation of the augmented control hierarchy of Figure 1(b). The physical instantiation will be presented later. The 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 2. By the nature of the 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.*

On the target machine, the target server T receives messages from the host server H and uses the messages to update shared target machine local variables which are then interpreted by other target processes. The Active Image

* The interactive/development facilities of the architecture will not be discussed here. A thorough discussion of this component can be found in [22].

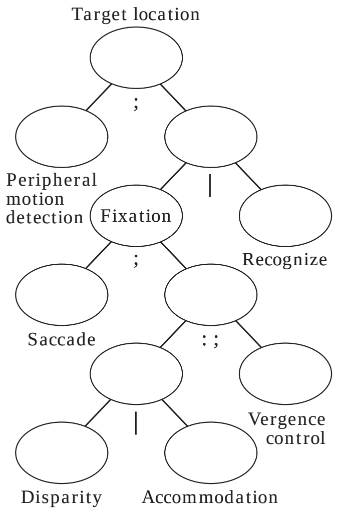
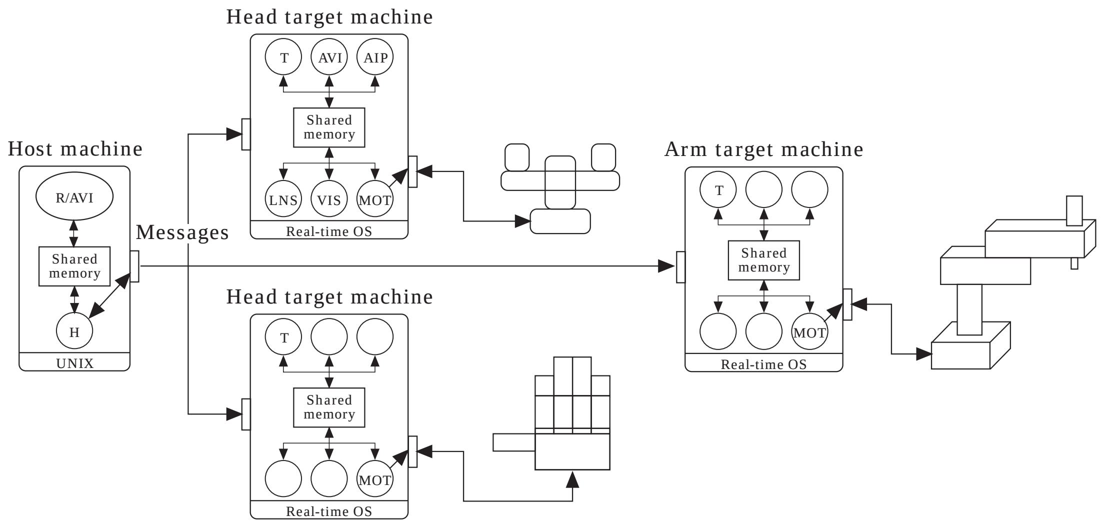


Figure 3. Target location parse tree.

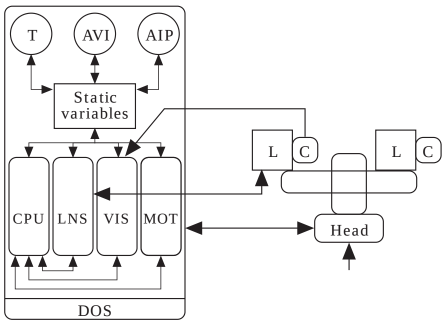
Processing process (AIP) is responsible for data filtering and fusion. The AV-integration process (AVI) is responsible for compensating system delays and composing higher-level attentive active vision processes into continuously running perception-action processes. The motion control process (MOT) provides servo control to the robotic actuators. Vision processes (VIS) carry out the pre-attentive image processing, while the (LNS) processes provide lens control.

AVI is implemented using the Robot Schemas (RS) notation introduced by Lyons [23]. 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 RS notation. For instance, *target location* is represented by the parse tree shown in Figure 3. The application of object recognition on an AVD such as a stereo robotic head is composed of the tasks of peripheral motion detection followed by fixation on the object creating the detected motion and recognition of the fixated object to determine if it is of interest. The target location example above illustrates the *RS* operators of sequential composition (;), concurrent composition (|) and synchronous recurrent



(a)



(b)

Figure 4. (a) Example system configuration; (b) detailed interconnection.

composition (:;). 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*.†

Our implementation of the interface facilities provides users with a standard interface to robotic manipulators, dextrous robotic hands, and AVDs. Each device is controlled by a real-time target machine configured and connected to the device. Figure 4(a) illustrates the configuration of the full architecture when multiple devices are being controlled (a robotic manipulator, a stereo robotic head, and a dextrous robotic arm), and Figure 4(b) illustrates the target machine configured for the stereo robotic head. Each device in our system is connected to a target machine similar to the one discussed above but configured for the particular device connected, as shown in Figure 4(a).

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

Approach

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) [24], Earliest Deadline First (EDF) [24], or time-line-based algorithms [25].

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:

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

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.* Each of the target systems implement RM to verify the feasibility of 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 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 earlier was used to obtain a fault-tolerant architecture, shown in Figure 5. This architecture provides fault tolerance at several levels. Using the language of [20], 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.

As mentioned above, the AVIs 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 main reason to use RM is that when many priority levels are needed, a hardware implementation of EDF is not possible.

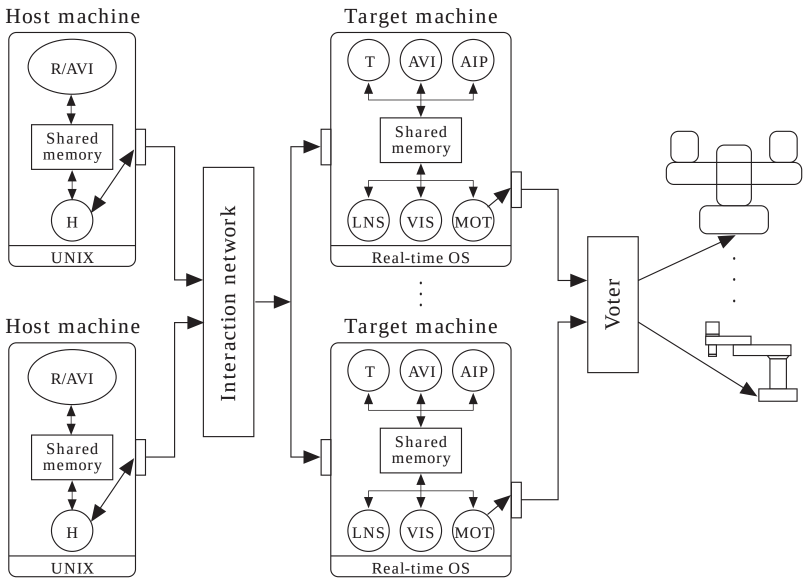


Figure 5. Logical system architecture.

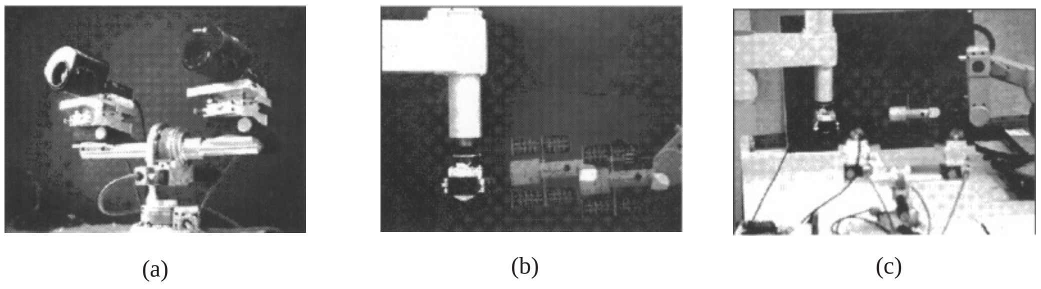


Figure 6: (a) Active vision head; (b) satellite and docking vehicle; (c) experimental configuration.

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. It is important to note that 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 or cheap to achieve.

Experimental Configuration

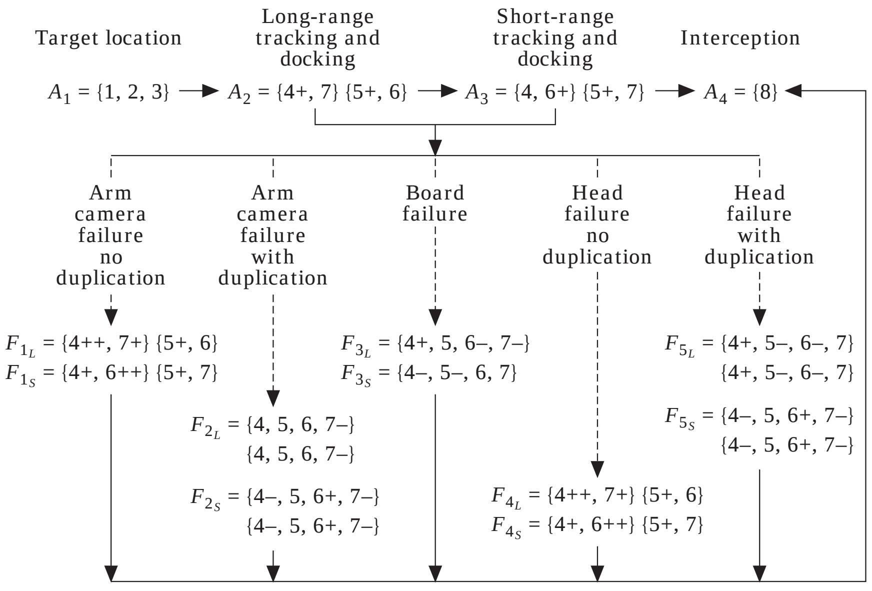
Our experimental configuration consists of an *active vision head* with four degrees of freedom (shown in Figure 6(a)), an AdeptOne robot arm (depicted as the docking vehicle on the left part of Figure 6(b)) with a camera mounted on it, and a satellite being held by a Scorbot-ER-9 robot arm (on the right part of Figure 6(b)). The entire configuration is shown in Figure 6(c).

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

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 7(a) 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 (hence the two sets of applications). Figure 7(b) summarizes the CPU utilization of the various scenarios.

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.


Target location
Long-range tracking and docking
Short-range tracking and docking
Interception
A_1 = {1, 2, 3}
A_2 = {4+, 7} {5+, 6}
A_3 = {4, 6+} {5+, 7}
A_4 = {8}
Arm camera failure no duplication
Arm camera failure with duplication
Board failure
Head failure no duplication
Head failure with duplication
F_{1_L} = {4++, 7+} {5+, 6}
F_{1_S} = {4+, 6++} {5+, 7}
F_{2_L} = {4, 5, 6, 7–} {4, 5, 6, 7–}
F_{2_S} = {4–, 5, 6+, 7–} {4–, 5, 6+, 7–}
F_{3_L} = {4+, 5, 6–, 7–}
F_{3_S} = {4–, 5–, 6, 7}
F_{4_L} = {4++, 7+} {5+, 6}
F_{4_S} = {4+, 6++} {5+, 7}
F_{5_L} = {4+, 5–, 6–, 7} {4+, 5–, 6–, 7}
F_{5_S} = {4–, 5, 6+, 7–} {4–, 5, 6+, 7–}

(a)

Scenario	CPU utilization
A_1	–
A_2	{47.4} {60.6}
A_3	{45.0} {63.0}
A_4	{10.0}
F_{1L}	{68.0} {60.6}
F_{1S}	{70.4} {63.0}
F_{2L}	{61.8}
F_{2S}	{66.1}
F_{3L}	{66.1}
F_{3S}	{63.0}
F_{4L}	{68.0} {60.6}
F_{4S}	{70.4} {63.0}
F_{5L}	{72.8}
F_{5S}	{66.1}

(b)

Figure 7. (a) Scenario applications and transitions. The recovery/contingency scenarios are labelled L if transition is from scenario A_2 and S if transition is from scenario A_3. (b) CPU Utilization of the various scenarios.

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 (70 cm 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 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

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

Appl	Functionality	Period	Exec time (ms)	% of CPU needed.
1	Motion detection	100.0	25.0	25.0%
2	Recognition/verification	–	100.0	–
3	Saccade	–	10.0	–
4–	Smooth pursuit	110.0	13.3	12.1%
4	Smooth pursuit	110.0	19.4	17.6%
4+	Smooth pursuit	110.0	30.1	27.4%
4++	Smooth pursuit	110.0	47.2	43.0%
5–	Trajectory estimation	750.0	100.0	13.3%
5	Trajectory estimation	750.0	162.0	21.6%
5+	Trajectory estimation	750.0	322.0	43.0%
6–	Docking	110.0	13.3	12.1%
6	Docking	110.0	19.4	17.6%
6+	Docking	110.0	30.1	27.4%
6++	Docking	110.0	47.2	43.0%
7–	Time-to-contact	100.0	5.0	5.0%
7	Time-to-contact	100.0	20.0	20.0%
7+	Time-to-contact	100.0	25.0	25.0%
8	Interception	–	10.0	10.0%

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 in which 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. $A_2 \Rightarrow F_{3L}$) 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. $A_3 \Rightarrow F_{3s}$) 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. $F_{3L} \Rightarrow A_2$), or {4–, 5–, 6, 7} $\Rightarrow$ {4, 6+}{5+, 7} (i.e. $F_{3S} \Rightarrow A_3$), which represents the reallocation of tasks.

In the case that a camera fails, there are several possible cases:

(i) Arm camera fails. 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 RM scheduling algorithm. Therefore, we must reconfigure the application allocation. In this case, we would use scenario {4+, 6 + +}{5+, 7}.

(ii) Arm camera fails, and application degradation and duplication are performed. In this case, the user wants the extra ability of tolerating more faults, such as the failure of one of the vision boards. Therefore, if failure occurs while in *long-range tracking and docking*, 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 docking procedure based on a sideways view, using the 6+ algorithm. Therefore, if failure occurs while in *short-range tracking and docking*, we enter the scenario {4–, 5, 6+, 7–}{4–, 5, 6+, 7–}. The scenario is duplicated in both vision boards allowing for tolerance of the failure of one of the vision boards.

(iii). Head camera fails. This situation is analogous to case 1; however, the head camera rather than the arm camera fails. In this case, since trajectory estimation needs to be carried out with guidance from the arm camera, more computationally intensive procedures must be executed (with only information from the arm camera). If the head 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 RM scheduling algorithm. Therefore, we must reconfigure the application allocation. In this case, we would use scenario {4+, 6 + +}{5+, 7}

(iv) Head camera fails, and application degradation and duplication are performed. This case is similar to case 2. The new duplicated scenario will be {4+, 5–, 6–,7}{4+, 5–, 6–, 7} if failure occurs: while in *long-range tracking and docking*. Similarly, the new duplicated scenario will be {4–, 5, 6+, 7–}{4–, 5, 6+, 7–} if failure occurs while in *short-range tracking and docking*. Here, time-to-contact is done less accurately through the arm-mounted camera in order to use the resources to execute the trajectory estimation. The docking process requires more computational power, since the input is received from the arm-mounted camera.

Experiments

The goal of our experiments is to verify the effectiveness of our architecture. To show this, we have implemented the experimental configuration described earlier. 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. Each module was analysed in terms of execution time and quality. With respect to fault tolerance, we check the accuracy of the different scenarios. We tested the scenarios for graceful degradation in terms of the accuracy of the modules involved and timing analysis for scenario switching. In the following subsections we will discuss our experiments in more detail. We begin with a detailed description of our experimental platform.

Experimental platform

All experiments conducted for this work were performed at the Intelligent Systems Laboratory at the Israel Institute of Technology – Technion (ISL). The architecture used in the ISL consists of two PC's: one a Pentium and the other a 486. Each PC contains an image processing card. The Pentium is used to control the TRH, while the 486 is connected to the camera mounted on the manipulator. Both PCs (target machines) communicate with each other and a Sun workstation (host machine) via ethernet. Figure 8 illustrates the Technion architecture.

The hardware consists of the following PC cards: (a) 486 processor – the arm PC is built around a 66 MHz 486 processor; (b) Pentium Processor – the head PC is built around a 90 MHz Pentium processor; (c) Technology 80 Model 28 Servo Controller – this card provides PID control to each of the four motors of the TRH head; (d) BarGold

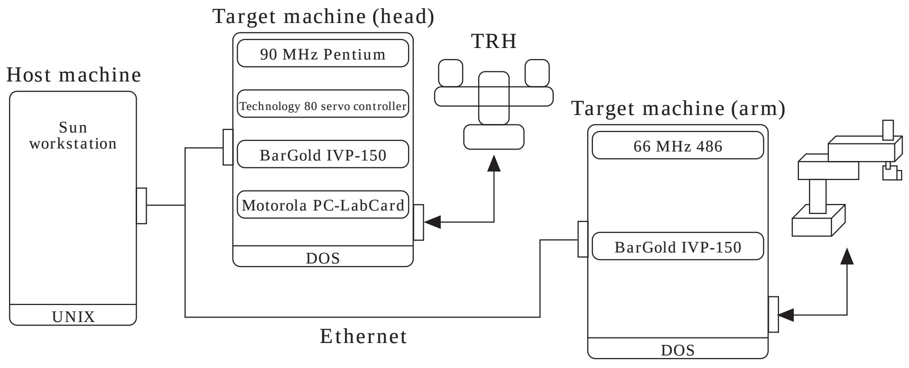


Figure 8. Experimental platform.

IVP-150 – these cards provide frame grabbing and image processing, and are capable of performing operations such as thresholding, convolution and image differencing at frame rates; (e) Motorola PC-LabCard: this card provides general purpose A/D and D/A conversions. It has been adapted to control the lenses of the TRH.

Techniques for smooth pursuit and docking

Smooth pursuit and docking are both based on motion analysis. In our experiments, we have implemented the following four motion analysis techniques which serve as the basis for both smooth pursuit and docking. The four techniques serve as the different levels of service for scenario-changing purposes.

(i) *Blob tracking*. Applications 4– and 6– are implemented based on blob tracking. Due to its simplicity and suitability for real-time implementation, blob tracking has been perhaps the single most commonly used technique for motion detection in tracking systems. Numerous works such as [26] have reported systems that are able to track a black or white blob. The source for many of these systems is a moving light such as a flashlight.

For evaluation of blob tracking, a black blob moving over a white background was used. The algorithm thresholds the input image to remove any spurious pixels, then computes the centroid of the blob. Blob tracking is the fastest of our implemented motion detection techniques. A sample can be computed in 13.3 ms.

(ii) *Edge tracking*. Applications 4 and 6 are implemented based on edge tracking. Edge tracking is similar in nature to blob tracking; however, rather than finding the centroid of the entire blob, the centroid of blob edges is found. In order to find the edges, an edge operator is used on the input image.

In our implementation of edge tracking, we first use the Sobel edge detector [27] to find image edges, then threshold the results to remove any spurious pixels. The centroid of the edges is then computed. A sample using edge tracking can be computed in 19.4 ms.

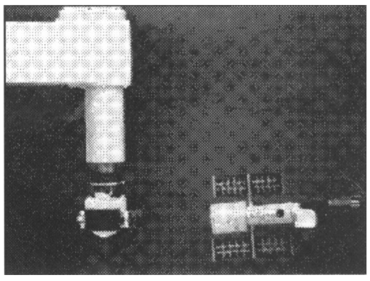
(iii) *Template matching*. Applications 4+ and 6+ are implemented based on template matching. For each template location, a similarity measure is computed, indicating how well the template matches the image at the current template location. The template location which provides the maximal similarity measure is selected as the location of the object in the image.

Differences in various template-matching techniques are usually found in the method used for computing the similarity measure. Several common techniques include the Sum of Squared Differences (*SSD*) technique and Normalized Cross Correlation. We use the SSD for computing the similarity measure in our implementation.

Our implementation of template matching makes use of the DSP on the BarGold image processing card, where a function efficiently implementing SSD correlation of a 15×15 pixel template with the entire 100×100 fovea yields the location with highest similarity. Using template matching, a sample can be computed in approximately 30.1 ms.

(iv) *Image differencing.* Applications $4++$ and $6++$ are implemented based on image differencing. Differencing segments the scene into static and moving regions, as only objects that have changed position between two consecutive images will have non-zero pixel values.

In our implementation of image differencing, the centroid of all pixels falling above a gray-scale threshold was computed over a two times sub-sampled fovea of 100×100 pixels in the difference image. Using image differencing, a sample can be computed in 47.2 ms.


(a)

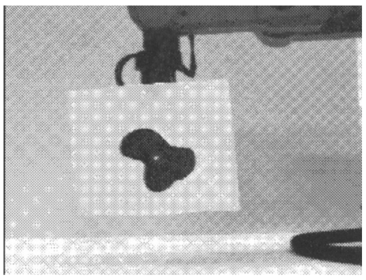

(b)

Figure 9. Sample of foregrounds and backgrounds used in evaluation experiments.

Motion analysis evaluation

The motion analysis techniques outlined above 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. Figure 9 illustrates two of these combinations. 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.

In each experiment, a robotic manipulator (holding the tracked object) effects horizontal translatory motion consisting of two motion segments: from the starting location to a location 200 cm to the right, and return to the starting location. The speed of the manipulator was set so that the entire 400 cm manipulator motion was completed in approximately 15 s.

Each module was designed to output tracking results in terms of the number of pixels that the object deviates from the image center. We recorded each motion sequence to videotape and extracted from the tape the pixel distance between image center and a central point on the moving object over the duration of the tracking. Tracking results of each module were classified as a success if some part of the object was located at the image center during the entire motion sequence) and a failure otherwise. Tracking module timing and module success ratios are given in the table of Figure 10(a). The plot of the data given in Figure 10(b) shows that using an algorithm with a slower execution speed leads to greater success.

The execution time gives the speed at which each module runs independently. However, in order to obtain comparable results, we eliminate variations in sampling rate which may influence the performance of the modules (often smaller execution times lead to improved performance). We normalized the sampling rate by running all modules each time while actuating the motors based on the results of only the particular module we were testing (this leads to the period of 110.0 shown in Table 1 for smooth pursuit and docking). Figure 11 shows that 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.

Techniques for time-to-contact

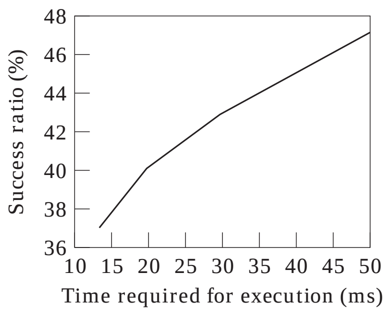
We have implemented two time-to-contact (TTC) modules. One module is divergence-based, while the other is model-based. It was shown in [28] that the rate of change of the area of an object, divided by the area, is proportional to the divergence. Using the divergence we can approximate the TTC. In the 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.

Time-to-contact evaluation

The time-to-contact modules were part of the configuration illustrated in Figure 6(c) (an AdeptOne robot arm with a mounted camera (docking vehicle), a satellite being held by a Scorbot-ER9 robot arm and a stereo robot head). Figure 12(a,c) shows the view of the satellite from the docking vehicle and head, respectively. The velocity of the motion was initially set at 5 cm/s. At a distance of 70 cm the velocity drops to 2 cm/s (in accordance with the scenario switch $A_2 \Rightarrow A_3$). All computations were performed in a fovea of 170×170 pixels. Both techniques were tested with a period of 100 ms. The divergence-based technique runs at 50 Hz, while the model-based method runs at 200 Hz.

Motion technique	Applications	Speed (ms)	Success ratio
Blob tracking	4–, 6–	13.3	37%
Edge tracking	4, 6	19.4	40%
Template matching	4+, 6+	30.1	43%
Image differencing	4++, 6++	47.2	47%

(a)



(b)

Figure 10. (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. (b) Plot showing that the slower the algorithm, the higher the success.

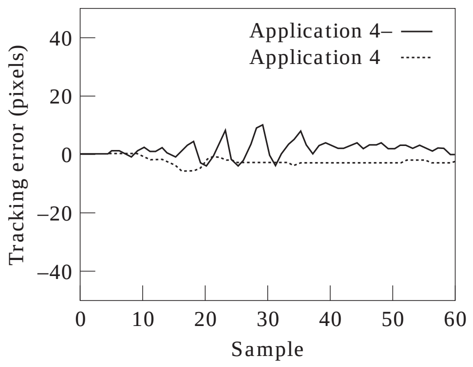


Figure 11. Tracking results from two motion modules.

Figure 13(a) shows the performance of the model-based technique (application 7–) over the entire satellite motion, while Figure 13(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 70 cm. As a result of this scenario switch, application 7 replaces application 7–. The scenario switch is shown in Figure 13(c). The discontinuity in the figure is a result of the new TTC stemming from the decrease in velocity from 5 cm/s to 2 cm/s*.

* Additionally, the model-based technique was tested with satellite velocity of 2 cm/s and the divergence-based technique with satellite velocity of 5 cm/s. The results were similar to those of Figures 13(a,b) and are not shown.

Table 2. Timing analysis for scenario switching

Scenario	A_2	A_3	F_{3L}	F_{3S}
A_2	.16	.16	.16	–
A_3	.16	.16	–	.002
F_{3L}	.159	.16	.16	.002
F_{3S}	16	.16	.16	.002

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. The diagonal in the table represents the time it takes to initialize the scenario. Measurements are given in seconds.

Conclusion

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. These capabilities are increasingly important in many areas such as space applications, nuclear plant control and security systems.

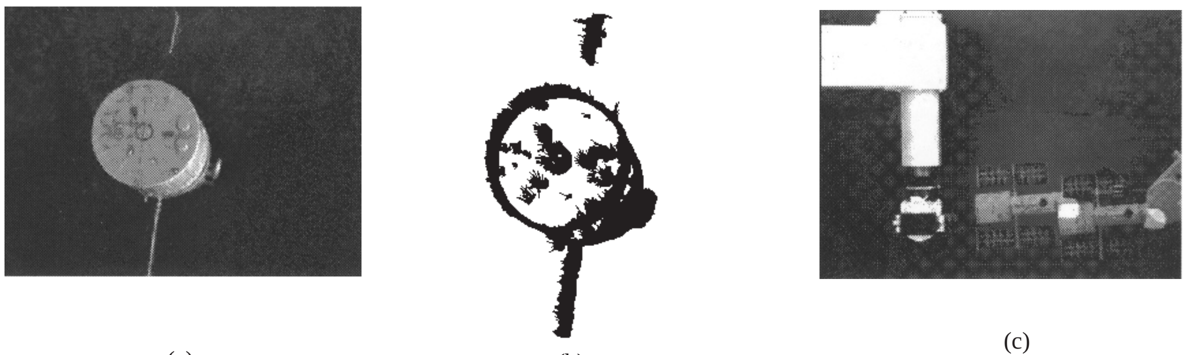


Figure 12. (a) View of satellite from docking vehicle; (b) optical flow for time-to-contact computations; (c) satellite motion viewed from head.

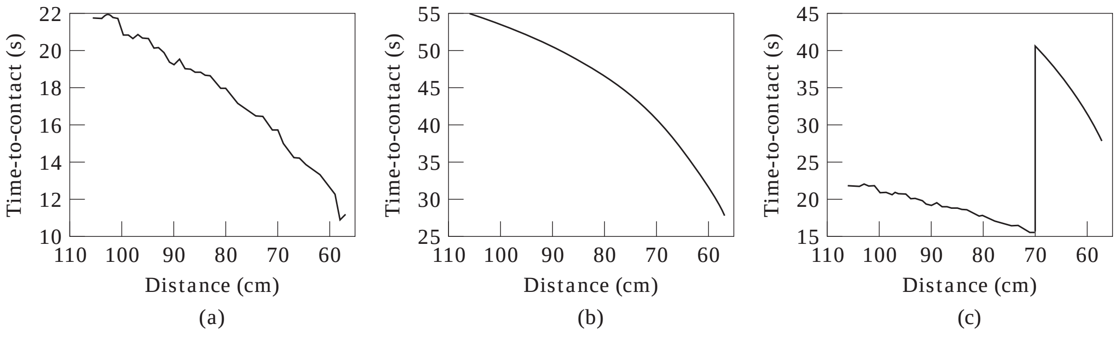


Figure 13. (a) Model-based TTC (application 7–) at 5 cm/s; (b) divergence-based TTC (application 7) at 2 cm/s; (c) scenario switch at 5 cm/s.

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. Due to our modular architecture users are able to execute the applications in the system in a dual mode of operation, namely simulation and experimentation (for rapid prototyping and testing, respectively).

In the experimental section, we presented our implemented platform and architecture. We describe four motion analysis modules and two time-to-contact modules, analysing them with respect to execution time and reliability. Additionally, we experimented with recovery latency (scenario switching) and showed that the overhead is quite low. These experiments provided information critical for scenario scheduling and allowed us to validate the results and performance of our system. The architecture is not limited to the particular scenario of our experiments, and can be easily adapted to many such problems. The algorithms we tested solve generic visual tasks such as tracking and time-to-contact, etc. We show how real-time demands impose changes in computational processes or forces their adjustment as a result of hardware faults.

References

1. Aloimonos, Y. Weiss, I., & Bandyopadhyay. (1987) Active vision. In: *International Journal on Computer Vision*, pp. 33–356.
2. Bajcsy, R. (1985) Active perception vs passible perception. In: *Proceedings of the Third IEEE Workshop on Computer Vision*, pp. 55–59, Bellaire, Michigan.
3. Ballard, D. H. (1991) Animate vision. *Elsevier Artificial Intelligence*, **48**: 57–86.
4. Crowley, J. L. & Christensen, H. I. (1995) *Vision as Process*. Springer-Verlag.
5. Birman, K. (1985) Replication and fault-tolerance in the ISIS system. In: *10th ACM Symposium on Operating System Principles*, pp. 63–78, WA. ACM.
6. Borg, A., Baumbach, J. & Glazer, S. (1983) A message passing system supporting fault tolerance. In: *9th AMC Symposium on Operating Systems Principles*, pp. 90–99, NH. ACM.
7. Cooper, E. (1985) Replicated distributed programs. In: ACM (ed.), *10th ACM Symposium on Oper Syst Principles*, pp. 63–78. ACM.
8. Schuette, M. A. & Shen, J. P. (1987) Processor control flow monitoring using signatured instruction streams. *Transactions on Computers*, C-**36**: 264–276.
9. Oh, S. K. & MacEwen, G. (1992) 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.
10. Bennet, S. & Virk, G. S. (eds) (1990) *Computer control of real-time processes*. London, UK: Institution of Electrical Engineering.
11. Kant, K. (1987) Software fault tolerance in real time systems. *Information Sciences*, **42**: 255–282.
12. Pollack, F. J. & Kahn, K. C. (1989) The BIIN mission critical computer architecture. In: *Proceedings of Workshop on Operating Systems for Mission Critical Computing*.
13. Svizienis, A. (1985) The N-version approach to fault-tolerant software. *IEEE Trans Software Engineering*, SE-**11**(12): 1491–1501.
14. Ghosh, S., Mosse, D. and Melhem, R. Implementation and Analysis of a Fault-Tolerant Scheduling Algorithm submitted for publication, 1994.
15. Kopetz, H. Damm, A., Koza, Ch., Mulazzani, M., Schwalbl, W., Senft, Ch., & Zainlinger, R. (1989) Distributed fault-tolerant real-time systems: the MARS approach. *IEEE Micro*.
16. Simmons, J. (1990) Concurrent planning and execution for a walking robot. Technical report, CMU.
17. Tso, K. S., Hecht, M. & Marzwell, N. I. Fault-tolerant robotic systems for critical applications. In: *Proceedings of the IEEE International Conference on Robotics and Automtion*, Atlanta, Georgia.
18. Visinsky, M. L., Cavellaro, J. R. & Walker, I.D., (1995) A dynamic fault tolerance framework for remote robots. *IEEE Transacionson Robotics and Automation*, **11**: 477–490.

19. Mossé, D. (1995) Creating resilient real-time applications. In: *20th IFAC/IFIP Workshop on Real-Time Programming*.
20. Mossé, D. (1994) Mechanisms for system-level fault tolerance in real-time systems. In: *International Conference on Robotics, Vision and Parallel Processing for Industrial Automation*.
21. Kligerman, E. & Stoyenko, A.D. (1986) Real-time euclid: a language for reliable real-time systems. *IEEE Transactions on Software Engineering*, pp. 941–949, September 1986.
22. Fayman, J., Rivlin, E. & Christensen, H. (1995) The active vision shell. Technical Report CIS 9510, Technion-Israel Institute of Technology.
23. Lyons, D. M. (1995) Representing and analyzing action plans as networks of concurrent processes. *IEEE Transactions on Robotics and Automation*, **9**: 241–256.
24. Liu, C. L. & Layland, J. W. (1973) Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment. *Jacm*, pp. 46–61.
25. Simmons, J. & Ramamritham, K. (1991) The spring kernel: a new paradigm for real-time systems. *IEEE Software*.
26. Clark, J. J. & Ferrier, N. J. (1988) Modal control of an attentive vision system. In: *Second International Conference on Computer Vision*, pp. 514–523, Tampa, Florida.
27. Ballard, D. H. and Brown, C. M. (1982) *Computer Vision*. Prentice-Hall.
28. Maybank, S. (1987) Apparent area of a rigid moving body. *Image and Vision Computing*, **5**: 111–13.