Experience with a Task Control Architecture for Mobile Robots

Long-Ji Lin
Reid Simmons
Christopher Fedor

CMU-RI-TR-89-29

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania

December 1989

This research was supported by NASA under Contract NAGW-1175.

Table of Contents

List of Figures

List of Tables

Abstract

This paper presents a general-purpose architecture for controlling mobile robots, and describes a working mobile manipulator which uses the architecture to operate in a dynamic and uncertain environment. The target of this work is to develop a distributed robot architecture for planning, execution, monitoring, exception handling, and multiple task coordination. We report our progress to date on the architecture development and the performance of the working robot. In particular, we discuss temporal reasoning, execution monitoring, and context-dependent exception handling.

1. Introduction

The principal goal of this work is to develop a distributed robot architecture to support robot planning, execution, monitoring, exception handling, and multiple task coordination. We have been developing such a robot architecture, called the Task Control Architecture (TCA) [15]. TCA is designed for controlling mobile robots that have limited computational and sensory resources, operate in uncertain, changing (but relatively benign) environments, have multiple goals, and have a variety of strategies to achieve goals and handle exceptions.

We have been developing TCA concurrently on two testbeds -- the CMU six-legged Planetary Rover [3] and the Heath/Zenith Hero 2000 mobile manipulator robot [12]. The CMU Rover project is an attempt to develop an autonomous robot that can survive, navigate, and acquire samples on the Martian surface. The Hero testbed is an indoor platform that has been used to drive the architecture design. The current capabilities of the Hero include collecting cups in the laboratory and recharging itself.

Our initial implementation on the Hero robot [12], which was developed in an ad hoc manner, had several shortcomings. It was slow and slack in reacting to environmental changes. It could not protect itself and recover from failures properly. It also could not change its focus to higher-priority tasks or respond to requests from human advisors. After re-implementing the testbed using mechanisms and functions provided by TCA, most of these shortcomings have been minimized. The robot is now faster and more robust. It can react to environmental changes in a reasonable time frame, and it has a variety of strategies to recover from failures.

The following are the capabilities that TCA currently supports.

- **Concurrent planning and execution.** Robots often take a significant amount of time in constructing plans. Since planning and execution are activities that often need different resources, both can occur concurrently. However, this concurrency sometimes needs to be constrained. In many cases, the robot must act on an incomplete plan and defer some specific decisions until more information can be acquired. On the other hand, to minimize risk to the robot, one might want to completely plan out a goal before executing any of its sub-commands.
- **Reacting to environmental changes.** To accomplish tasks, and even to survive, the robot must be reactive. It must always be aware of environmental changes, and respond to them appropriately and in a timely manner. Some environmental changes invalidate current plans, while others may demand the robot to change its focus completely.
- **Error recovery.** In complicated, changing environments, failures are bound to occur. When they do occur, the robot must change its plan to meet the new situation. Error recovery is often context-dependent, that is, the same failure may have to be handled differently, depending on the robot's intentions. Since in a benign environment, the failed plan is often close to being correct, it is desirable for the robot to be able to fix and re-use the problematic plan, instead of always replanning from scratch.
- **Coordinating Multiple Tasks.** With many simultaneous goals but limited resources, the robot must be able to dynamically prioritize and schedule its various tasks based on their urgency, relative costs, likelihoods of success, etc. Currently, only simple-minded strategies can be specified using TCA, but we envision taking a more knowledge intensive approach in the near future.

Various TCA mechanisms have been developed to support these capabilities.

- **Distributed processing.** TCA is a distributed architecture with centralized control. A robot system using TCA includes a central control and a number of concurrent, application-specific

processes. We believe that a centralized control architecture facilitates the coordination of multiple complex robot behaviors, while the distributed processing allows for concurrency in planning, execution, and perception.

- **Resources.** TCA provides a mechanism to schedule the use of the robot's limited computational and physical resources. A task is automatically queued by TCA until the needed resources are available. Resource reservation, together with temporal constraints (see below), provide synchronization mechanisms to control distributed robot systems.
- **Task trees and temporal constraints.** In TCA, planning and execution are separate activities and can be performed concurrently. The interleaving of these activities can be constrained by imposing temporal constraints among the planning and achievement times of subgoals. TCA explicitly maintains the goal/subgoal hierarchies, called *task trees*. Task trees, together with the temporal constraints, are TCA's representation of plans.
- **Concurrent monitors.** Concurrent monitors enable the robot to watch for environmental changes in parallel with normal task execution. Because task execution and monitoring occur concurrently, the performance of tasks will not be (significantly) slowed down, while still enabling environmental changes to be detected as early as possible.
- **Exception handling.** TCA provides a general mechanism for handling planning time failures, execution time errors, and contingencies. The robot implementor can specify different strategies for handling the same exception in different contexts. One benefit of having this mechanism is to allow the user to separate robot behaviors for normal situations from these that handle failures or contingencies. In this way, complex robot behaviors can be developed incrementally, and exception handling can be flexibly defined. At present, the mechanism is still under construction but some primary results have been obtained.

Table 1-1 summarizes the supporting relationships between the TCA mechanisms and desired robot capabilities. A mark "X" in an entry of the table indicate that the mechanism in that column is used to support the capability in that row. Note that although synchronization by itself is not a capability needed by robots, it plays an important role in the distributed environment of TCA.

Table 1-1: The supporting relationships between mechanisms and capabilities

Mechanisms / Capabilities	Distributed Processing	Resources	Task Trees & Temporal Constraints	Concurrent Monitors	Exception Handling
Synchronization		X	X		
Concurrent Planning & Execution	X		X		
Reacting to Changes	X			X	X
Error Recovery			X		X
Coordinating Multiple Tasks		X	X		

The rest of this paper presents the Hero robot system, the Task Control Architecture, and their performance. Section 2 describes the hardware setup of the system and gives a scenario to illustrate how the Hero robot performs tasks. Section 3 discusses the various mechanisms of TCA. Section 4 describes the robot system in detail. Performance of the robot and TCA is evaluated in Section 5. Comparisons with related work are given in Section 6. Finally the paper is concluded in Section 7.

2. Scenario

Our mobile manipulator robot, the Heath/Zenith Hero 2000, is a commercially available wheeled robot with a two-finger hand (see Figure 2-1). The robot operates in an unstructured laboratory, which is observable through a ceiling-mounted camera (see Figure 2-2). The Hero robot has three sonar sensors: a rotating sonar on top, a forward-pointing sonar fixed to its base, and one mounted on the robot's hand which can be repositioned relative to the body. In addition, the robot has a battery charge level sensor, a rotating light intensity sensor, and touch sensors on the fingers. Using existing vision software [10], we developed a 2D vision subsystem for the ceiling camera. We also developed algorithms for navigation and manipulation in the indoor environment.

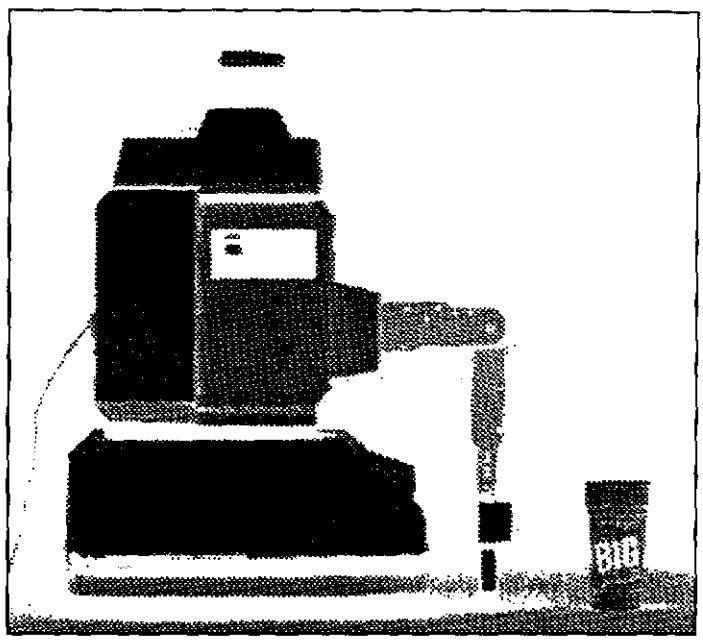

Figure 2-1: The Hero 2000 Robot

When the system is started up, the robot is given several high-level goals, including (1) collecting cups discovered on the lab floor and placing them in a receptacle, (2) avoiding obstacles, and (3) recharging its battery when necessary. The rest of this section presents a scenario to illustrate how the robot achieves and coordinates these goals.

For the cup collection task, the robot monitors its 2D vision map for the appearance of cups on the floor. An asynchronous perception process continually takes a picture and updates a world map. Once a new map is built, the robot scans the map to find cup-like objects. In this scenario, two cup-like objects are spotted, and the system sets up two *cup-collection* goals and temporally orders them so that the closer object will be explored first.

The robot then plans and executes a path to the first object. While moving, it monitors for obstacles in its path. A monitor, whose temporal extent continues until the object is picked up, is created to ensure that the target object does not disappear (e.g., someone else may pick it up). Upon arriving near the object, the robot uses its wrist sonar to measure the height and width of the object and matches them against its cup models. If a satisfactory match is found, the robot plans and executes actions to pick up the object. In parallel with measuring and picking up the object, the robot uses its overhead vision map to pre-plan a path to the receptacle so that a path plan is ready for execution when the cup is picked up. The

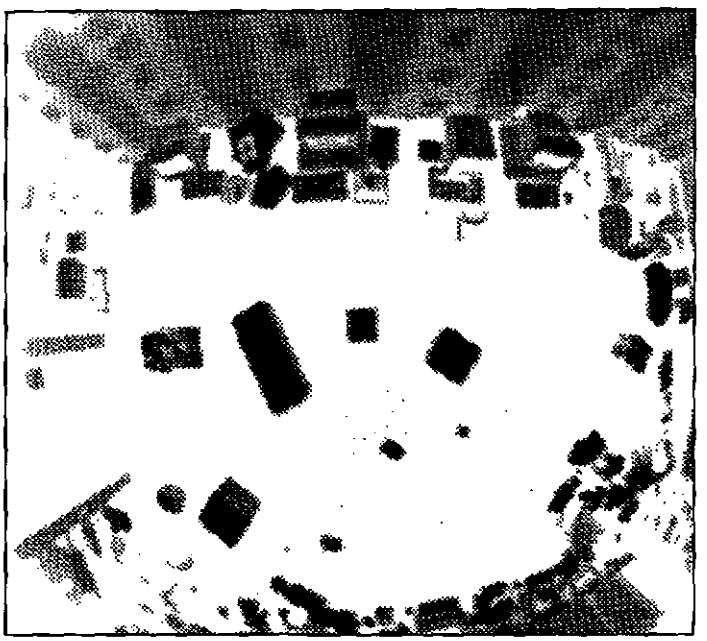

Figure 2-2: Overhead View of Laboratory as Seen by Robot

robot then uses the plan to navigate to the receptacle, where it deposits the cup.

Next, the robot attends to collecting the other object. While moving toward the object, the robot notices (from either its overhead vision or its sonar sensors) that an object appears in into its path. The robot stops immediately and waits to see if the object will move away soon. If the obstacle does not move, the robot plans a detour by modifying the blocked path plan. If no detour can be found, the robot replans a path from scratch. If still no path can be found, the robot abandons this cup-collection goal.

In this scenario a detour is found, so the robot continues to navigate to the object. The robot finally arrives near the object and starts measuring it. At this point, the battery charge monitor notifies the robot that its battery charge is getting low. Based on the simple-minded strategy: "if the robot has arrived near the object complete the task before going over to recharge", the robot creates a *recharge* goal with temporal constraints indicating that the new goal will be attended to after the cup-collection goal is achieved or aborted. The robot continues and subsequently discovers that the object is not a cup at all. It gracefully terminates all ongoing and pending activities and monitors that were set up for collecting the object, and then it chooses to pursue its next goal, which is the *recharge* goal.

3. The Task Control Architecture

TCA is designed to implement capabilities we believe to be necessary for autonomous robots. TCA is a distributed architecture with centralized control. An application of TCA includes a central process and a number of concurrent, application-specific processes, called *modules*. Communication occurs via coarse-grained message passing between modules, with all messages being routed through the central process.

To facilitate experimentation with different control schemes, TCA is built as a layered system so that an implementor can choose which layers to use -- higher layers provide more functionality specific to robot control, but lower layers provide flexibility to implement alternative control schemes.

At present, the implemented layers include:

- Communication layer that supports distributed processes under centralized control;
- Behavior layer for querying the environment, specifying goals, executing commands, and altering the robot's internal state
- Resource layer for allocating and managing physical and computational resources;
- Task management layer for building hierarchical plan structures and specifying temporal constraints between planning and execution of various goals in the plan;
- Monitor layer for concurrently monitoring user-selected aspects of the robot's external and internal environments;
- Exception handling layer for specifying context-dependent strategies for handling plan failures, execution errors, and environmental changes.

In addition, other layers to support multi-task coordination and user interaction are planned.

3.1. Communication Layer

The base layer of functionality provided by TCA is the sending and receiving of messages between modules. Modules can be written in different languages (currently both Lisp and C are supported) and run on different machines (using the UNIX TCP protocol). In essence, TCA provides a simple *remote procedure call*(RPC) interface from a caller in one module to a procedure in a possibly remote module. The main difference between typical RPC implementation and TCA is that the central control determines which module handles messages and in what order they are handled.

A potential problem with centralized control is that the central process may become a bottleneck. Experimentally, a round-trip time for messages of under 10K bytes is about 50 milliseconds. Since this time is small compared with the time taken by image processing, planning, and the robot's actuators, the centralized control has not been a problem on our current testbeds. Besides, the potential bottleneck problem can be overcome by using high-speed hardware (e.g., the Nectar [2]) and adhering to some conventions, such as using coarse-grained behaviors to limit the amount of module-to-module communication.

3.2. Behavior Layer

TCA provides several types of primitive building blocks needed to construct robot behaviors. The primitive behaviors are implemented as different classes of messages, built on top of the communication layer. The classes differ mainly in their control flow. For example, query messages block the user's code until a reply is received, while goal and command messages are non-blocking and report success or failure directly to the central control.

- **Query messages** are requests to provide information about the external or internal environment, such as obtaining a world map or determining the robot's dead-reckoned position.
- **Goal messages** are intended to support top-down, hierarchical planning. A typical response to a goal message would be to issue other (sub)goal and/or command messages based on the results of planning. Unlike queries, goal messages are asynchronous and non-blocking. That is, the central control may queue the goal until resources become available; in the meanwhile, the module sending the goal message can continue. The rationale is that non-blocking goal messages give the implementor greater flexibility in controlling the achievement of goals

(e.g., interleaving planning and execution).

- **Command messages** are used to execute actions. Like goal messages, command messages are asynchronous and non-blocking. Distinguishing goal from command messages is done mainly for interleaving planning and execution.
- **Constraint messages** provide a way to alter the robot's internal state. For example, constraint messages can be used to add expectations about its future behaviors.

3.3. Resource Layer

It is crucial for an autonomous agent to effectively allocate its limited resources in order to satisfy its goals. The robot must detect when tasks need competing resources, and must prioritize and schedule tasks when conflicts occur. In TCA, a resource is an abstract entity that is used to manage the handling of messages. A resource may be associated with a computational entity, such as a module, or with a physical entity, such as a motor or camera.

Resources are created with a capacity - the number of messages the resource can handle simultaneously. A message received by the central control is queued until the resource that handles the message has available capacity. Currently, messages to the same resource are handled in FIFO order, subject to the temporal constraints imposed by the task management layer.[1]

Sometimes, a module might need control over a resource for some period of time, particularly one associated with a physical item. For example, if a vision module is acquiring an image, it might want to ensure that the robot does not move during that period. To facilitate this, TCA includes mechanisms for reserving resources, in effect, preventing other modules from utilizing the resource until the reservation is explicitly canceled. Resource reservation is one of the synchronization constructs in TCA.

3.4. Task Management Layer

The task management layer provides mechanisms for organizing sets of messages into hierarchical *task trees* (see Figure 3-1). For each goal, command, or monitor message sent by a module, TCA adds a node to the task tree as a child of the node that issued the message. The resulting tree is an execution of graph of messages used to complete a given task. In addition, facilities have been developed for tracing and manipulating the task tree, such as killing off subtrees, suspending them, and adding new nodes. These facilities will provide functionalities needed by some of the higher layers, such as the exception handling layer (see Section 3.6) and the planned multi-task coordination layer.

Another important purpose of this layer is for scheduling tasks. The layer contains a general facility for reasoning about time. In TCA, by default planning and execution can occur concurrently. Interleaving of planning and execution can be constrained by imposing temporal constraints on the planning times of goals and achievement times of goals, commands, and monitors. For example, a module might specify that the achievement time of G1 precedes that of G2, but the planning time of G2 precedes that of G1 (e.g., first achieve pick up the cup, then bring it to the receptacle, but plan the route to the receptacle before planning how to pick up the cup). Similarly, a module might constrain a goal to be completely planned before any of its sub-commands can start being achieved.

---

[1] We plan to add more sophisticated scheduling mechanisms in the future.

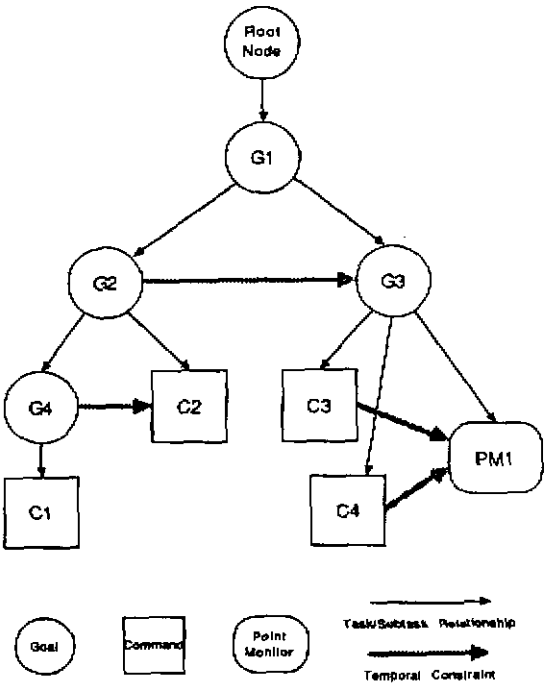

Figure 3-1: Sample task tree

The mechanisms for reasoning about temporal constraints are based on the *Quantity Lattice* [13], an arithmetic reasoning system, that integrates *relationships, arithmetic expressions, qualitative* and *quantitative*[2] information to perform a wide range of common arithmetic inferences. In TCA, it is used to maintain a consistent partial order of time points and to answer queries about relationships between time points and about the durations of intervals.

With the temporal mechanisms provided, robot implementors can formulate a fairly wide range of different constraints to take advantage of concurrencies in the distributed environment of TCA. Together with resource reservation, the temporal constraints provide synchronization mechanisms to control distributed robot systems.

3.5. Monitor Layer

To react to environmental changes, robots must first be able to monitor the environment and detect changes in time. Although in the real world many things may go wrong at any time, robots with limited sensory resources, such as ours, cannot afford to monitor everything that goes on in the environment. The monitor layer provides mechanisms to monitor user-selected aspects of the environment and report detected changes to the central control for handling. Monitors in TCA run concurrently with normal task execution. For example, the Hero robot attends to the cup collection goal while monitoring for obstacles and its battery charge.

A monitor specifies the condition to be monitored, and the time, relative to other messages, when monitoring is to take place. When the condition holds, a typical action would be to send an **exception message** to the central control, which will decide what to do based on the environment and context in

[2]The quantitative reasoning capability of the Quantity Lattice is not yet utilized by TCA.

which the exception occurred (see Section 3.6).

Two classes of monitors are implemented: *point monitors* and *interval monitors*. Point monitors, which test the monitor's condition just once, are useful for checking static, execution time conditions, such as checking the pre-condition or post-condition of a command or goal. Interval monitors, which have a temporal extent, are useful for checking for environmental changes over time.

TCA has two variations of interval monitors: *polling* and *demon monitors*. Polling monitors implement synchronous polling of conditions at a fixed frequency, while demon monitors implement asynchronous demon-invocation. For instance, the battery monitor of the Hero robot, which is a polling monitor, periodically checks the battery charger and raises an exception if a low charge is detected. The cup appearance monitor, implemented as a demon monitor, is invoked whenever a world map is updated by the asynchronous perception process, and checks the world map for cup-like objects, raising exceptions if such objects are found.

Monitors can also be used to construct conditional plans. For instance, suppose there are two strategies to achieve goal G, but we do not know in advance which one will be applicable. We can set up a monitor to check the environment and choose the appropriate strategy at execution time.

3.6. Exception Handling Layer[3]

Exceptions can be divided into three classes, according to the ways they are detected.

- failures detected in planning (e.g., no path to the cup);
- errors detected in executing commands (e.g., wheel slippage);
- contingencies detected by monitors (e.g., low battery charge).

TCA employs the same mechanisms to handle the three different types of exceptions.

Exception handling is often context-dependent: the same exception might need be handled differently, depending on the environment and where in the plan the exception occurs. For example, a wheel blockage is a failure if it is detected when the robot is navigating in an open space. But it could be a signal of a successful docking if the robot's goal is to dock on the charger. To facilitate context-dependent exception handling, TCA supports mechanisms for associating exception handlers with contexts at planning time and automatically invoking the handlers when exceptions are raised. Various utilities are also provided to enable handlers to fix problematic plans.

The context of an exception handler is established by attaching the handler to a task tree node. This association is done dynamically as the task tree is created. When an exception is raised, TCA searches up the task tree, starting from the node where the exception arose, to find a handler specific to that exception. The first matched handler is then invoked to handle the exception.

Exception handling is achieved by editing the task tree, for example, by deleting part of it and inserting some new nodes. The exception handlers can use the task tree operations provided by the task management layer to access, scrutinize, and then modify the task tree. Modifications to task trees may include terminating or suspending the execution of subtrees, and adding new nodes to the task tree, which

---

[3]Currently, only the framework of the exception handling layer has been implemented, and various supporting mechanisms are still under construction.

is then expanded using the normal TCA mechanisms. To illustrate, Figure 3-2(a) shows a situation where a *battery charge* monitor is set up and the robot is actively attending to the *cup-collection* goal. When the monitor detects a low battery charge, the *low battery charge handler* attached to the root node is chosen to handle it. After checking the battery charge and the progress of the cup collection, the handler decides to recharge first and finally ends up with the situation in Figure 3-2(b), where the monitor has been canceled, the *cup-collection* goal has been suspended, and the *recharge* goal has been added and become the current goal.

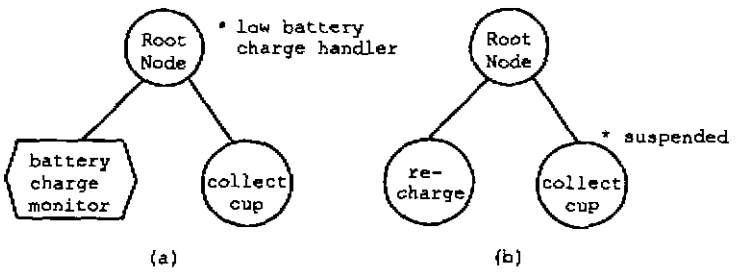

Figure 3-2: Exception handling

If an exception handler finds it cannot actually handle the situation, it can raise an exception itself. When the central control receives an exception from an exception handler, the search for a capable handler is resumed, starting from the node where the previous handler was found and searching up the task tree. This process is repeated until the exception is successfully handled. As a catchall, TCA attaches a general exception handler to the root node of the task tree. When invoked, this general handler simply deletes the failed task along with all its subtasks.

This TCA approach to exception handling is efficient. First, the invocation of exception handlers is fast, because only a simple search on the task tree is involved. Second, TCA allows a problematic plan to be fixed and re-used as much as possible. For example, when moving obstacles appear unexpectedly, the Hero robot first waits for obstacles to move away. If they do not move away, it tries to plan a detour by modifying the blocked path plan. If no detour is found, a new path is planned from scratch. Only if no path is found is the task terminated.

4. The Hero Robot System

The Hero robot system, which uses TCA, presently consists of five modules plus the central control (see Figure 4-1). In this section, we describe the functionalities of the modules and how they interact with each other.

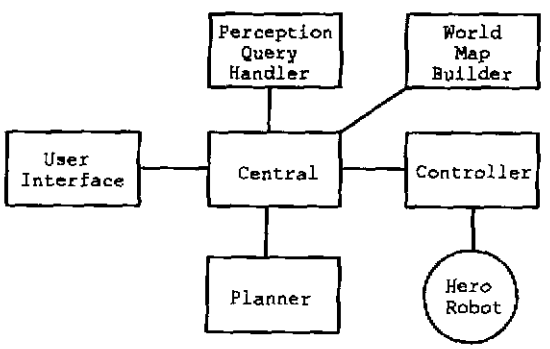

Figure 4-1: Organization of the robot testbed

Controller. This module, which controls the robot via either a radio link or an RS232 cable, executes navigation commands (e.g., turn, move) and manipulation commands (e.g., raise arm, open grippers). It also handles queries that involve using sensors on the robot, for example, reading the battery charge level, and measuring the height of an object using the wrist sonar.

The Controller also keeps track of the robot's trajectory and handles trajectory queries. Because of the control error, the uncertainty about the robot's position will grow over time. The Controller utilizes a covariance matrix representation [16] to model the control error, and compounds the uncertainty whenever the robot moves or turns. This uncertainty information is primarily used by the Perception Query Handler to determine the likelihood of hitting obstacles in the course of navigation.

We also implemented reflexive guarded move commands directly on-board the Hero. These give the robot a higher degree of reactivity than could be gotten from centralized control. While the robot is moving or turning, the on-board CPU detects wheel slippage and blockage by monitoring the motor encoders. At the same time, the sonar sensors are used to detect obstacles in the robot's trajectory. In both cases, the reflex action is to stop the robot immediately, stabilizing it. Then the Controller signals a failure so that the system can rectify the situation using the exception handling mechanisms.

World Map Builder. This module continually takes and processes images of the lab (every 20 seconds or so), and updates a world map, which is then forwarded to the Perception Query Handler. We have found that this asynchronous process has substantially increased the performance of the robot compared with our previous system. For example, since a relatively up-to-date world map is always available, the robot does not need to wait for processing an image in order to find a cup-like object or to plan a path.

To identify the robot in the image, the World Map Builder first gets the robot's dead-reckoned trajectory from the Controller. Based on the trajectory and other information such as the size of the robot, the robot region can often be distinguished from other object regions. Two failures, however, can be encountered. First, the robot may not be successfully spotted, because the robot region, for example, overlaps another visual region. This failure is handled by taking an image, moving the robot a few inches, taking another image, and comparing the differences in the images to spot the robot. The second failure occurs when the light in the lab is turned off. This exception is handled by asking humans to turn on the light or going to sleep (i.e., turning off the power to all circuitry except the memory) if no help is secured.

Perception Query Handler. The Perception Query Handler provides three kinds of functionality. First, it updates the world map upon receiving a new map from the World Map Builder. Second, it handles perception demons. When a new world map is received, perception demons are invoked to check conditions that they monitor. Presently there are two kinds of demons that can be set up - cup appearance monitors and object monitors (for checking if an object remains at a specified position on the floor).

The third task of this module is to handle perception queries, including

- calculating the vicinity of an object in order to approach it,
- checking if a path is clear, based on uncertainty reasoning,
- reducing the uncertainty about the robot's location and orientation by using vision.

As mentioned previously, the Controller explicitly models the uncertainty of the robot's status. When the robot is executing a path plan, the Perception Query Handler, given the uncertainty information, would be

asked to determine (1) if the path is clear, (2) if yes, how far the robot may safely proceed along the path before the uncertainty cone overlaps object regions (see Figure 4-2). If the uncertainty has grown to the extent that collisions with obstacles are possible, the Perception Query Handler uses vision to reduce the uncertainty. To do this, it first takes a picture of the robot and calculates the robot status (including visual uncertainty) based on properties of the robot's shape and internal model of sensor uncertainty. A new robot status is then obtained by merging the observed and expected status [16].

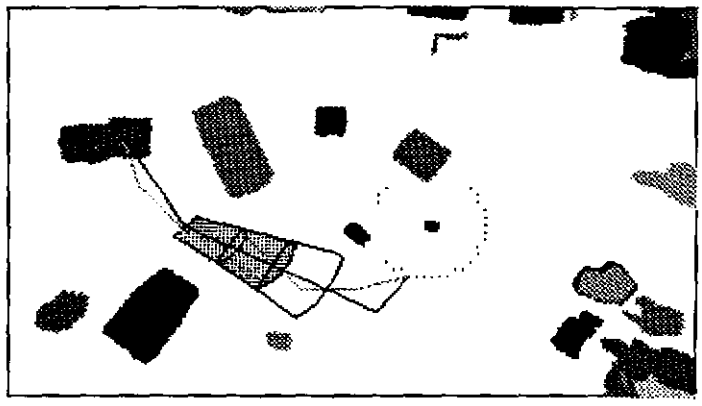

Figure 4-2:
Interpreted Version of the Image from Figure 2-2 with Planned Path and Uncertainty Cone. The brightened line shows the final computed path to a cup-like object, while the dimmer line is the original path before optimization. The shaded area in the uncertainty cone indicates how far the robot may safely proceed.

Planner. At present most of the navigation and manipulation planning is done in this module. The Planner has a collection of procedures, each of which is intended to achieve a goal. When executed to achieve goals, the procedures typically send queries, create subgoals, issue commands, set up monitors, specify temporal constraints, and/or associate exception handlers with contexts.

As an example, the procedure for handling the cup collection goal does the following:

1. Adds *approach object* goal. The first step is to navigate to the vicinity of the target object. In the course of navigation, the robot models uncertainty and watches out for obstacles.
2. Sets up *object monitor*. This monitor watches for the disappearance of the target object. Temporal constraints are added to indicate that the monitor starts from the beginning of the cup collection goal and ends at the beginning of the *grasp cup* goal (see below).
3. Adds *servo to object* goal. Once arriving near the object, the robot utilizes its wrist sonar to estimate its distance and orientation relative to the object. This information is used to compute the locomotion commands to reduce the differences between the estimated and desired distance and orientation. To overcome sensing and control errors, this goal is re-generated recursively until the differences are within acceptable limits. This recursive implementation makes it possible to break the time-consuming servoing loop for handling contingencies.
4. Adds *identify object* goal to measure and classify the object.
5. Adds *grasp cup* goal. If the object is a cup, it is grasped by a procedure specific to that cup. A point monitor, which utilizes the base sonar, is set up for checking if the grasping

succeeds.

6. Adds *approach receptacle* goal. Once picked up, the cup is brought to the receptacle. However, temporal constraints are imposed so that the path planning can begin once the robot arrives near the cup.
7. Sets up *holding monitor*. This interval monitor periodically reads the sensors on the fingers to make sure that the cup does not drop on the way to the receptacle.
8. Adds *deposit* command to drop off the cup in the receptacle.
9. Associates appropriate exception handlers to various task tree nodes.

User Interface. Presently the User Interface merely allows the user to enter commands, add goals, and set up monitors. Facilities for supporting a friendly user interface are being planned.

5. Performance

Our experience with the testbed shows that TCA is a helpful tool for building robot behaviors.

- TCA is easy to use and programs developed under TCA are usually easy to extend and modify. This is partly because TCA encourages modularity of programs. For example, normal robot behaviors, monitors, and exception handling can be developed separately.
- TCA provides a fair amount of expressive power to facilitate implementing complex robot behaviors. For example, TCA makes it easy to specify and control the interleaving of planning and execution, concurrent monitors, and exception handling.

Due to its deliberative nature, TCA cannot be used to implement low-level reflex behaviors that demand sub-second responses to environmental changes. To minimize the interval between the time an exception is detected and the time the exception handler gets executed, the implementors themselves must adhere to a principle: each of the robot's primitive actions must be designed to finish in a small time frame. In other words, a time-consuming action must be repeatedly divided into smaller ones, so that each does not take much time. The reason is that when an exception is raised, the chosen exception handler might be blocked by other ongoing primitive actions, because of resource conflicts. If so, the handler must wait for these actions to finish. Guaranteed reactivity is an interesting research area and we plan to investigate it in the near future.

Roughly speaking, the robot system described above is quite successful in surviving, collecting cups, and maintaining battery charge. It typically takes about 3-5 minutes to collect a cup, depending on the difficulty of individual tasks (e.g., smaller cups usually demands more time). If a cup is placed away from the perimeter of the visual view and not occluded, the robot can locate and collect it most of the time. Although the vision subsystem can be easily fooled by small non-cup objects (e.g., small box, sneaker), those objects are usually identified as non-cups by the sonar sensors when the robot approaches the objects (but they can result in considerable wasted time).

The robot system is about twice as fast as the previous sequential version. This is mainly because the world map is updated by an asynchronous process; this is a big win, because image processing takes much time. Another speed-up results from concurrent monitors and concurrent planning and execution.

The robot system is also relatively robust compared with the previous version. This is mainly because the concurrent monitors enable exceptions to be found early and the robot has a variety of strategies for handling exceptions. It is also helped by the reflexive guarded commands and their integration into the

TCA mechanisms.

The robot, however, is still susceptible to dangers. These dangers mainly arise from the robot's inability in sensing. For example, the robot has no sensor to detect imminent arm collisions and prevents them in advance. The vision processing is slow, so the robot might use out-of-date information and make wrong decisions. Although these problems can be minimized (but not overcome) by adding more sensors and using faster hardware, that is not the purpose of this work.

6. Related Work

An alternative approach to building reactive and robust robots is that taken by the subsumption architecture [4]. The main features of this approach are (1) hard-wired, layered robot behaviors, (2) no explicit internal model of the world, (3) no explicit representation of goals and plans, (4) no central control, and (5) continual monitoring. Many of these characteristics are shared by some other approaches, such as [1] and [11]. In contrast to these architectures, TCA has a centralized control and makes the notion of goals explicit, allowing the robot to reason about them. These differences make TCA more flexible in coordinating complex robot behaviors. The use of explicit plan representations enables TCA to pre-plan for the future, not just figure out "what to do next". TCA advocates selective monitoring, because sensors are often scarce resources and the use of them should be carefully scheduled. These differences result in two architectures with very different capabilities [6]. While the subsumption architecture is good at handling low-level sensor and effector actions (e.g., car chasing), it is not yet clear how complex behaviors (e.g., planning, exception handling) can be coordinated in the architecture. On the other hand, while with TCA fairly complex behaviors have been realized on the Hero robot, it is not well-suited to handling low-level reflex activities. Rather than competing architectures, however, it is reasonable to combine the strengths of both approaches, for example, by using the subsumption architecture for reflexive control, which talks to TCA for higher-level control. In fact, our experience with the guarded move commands (see Section 4) suggests that this might be a promising way to implement robust, intelligent robots.

The Procedural Reasoning System (PRS) [7] consists of four main components: a database of beliefs about the world, a goal stack, a library of procedural plans, and an interpreter. PRS is similar to TCA in several aspects. For example, both are concerned with combining planful, reasoned behaviors with reactivity. The goal stack and procedural plan representation used in PRS is similar to our task tree structure plus temporal constraints. The main difference between the two systems is that PRS is more concerned with reasoning and planning, while TCA mainly focuses on the execution, monitoring, and exception handling.

The Reactive Action Package (RAP) system [5], which is very similar to PRS, is another work which addresses reactivity and adaptive execution of plans. Like TCA, the RAP system provides various mechanisms for supporting resource reservation, temporal constraints, monitoring, and exception handling. The RAP system, which is a sequential system, is based on the idea of situation-driven execution, much like the subsumption architecture. This viewpoint is different from that of TCA. While supporting reactivity, TCA still allows the robot to plan for the future. For example, the Hero robot can measure the potential cup, monitor its battery charge, and pre-plan the path to the receptacle concurrently. Both systems also differ in the ways exceptions are handled. When exceptions are raised, the RAP system examines the context at run-time to find the appropriate method for re-achieving the failed task, while in TCA only a simple search on the task tree is needed.

The exception handling mechanisms of TCA are similar to those in some programming languages such as Ada [9] -- when an exception occurs, program execution is transferred to the exception handler with a matched name that is closest to the exception point in the context (i.e., the runtime call-stack in Ada or the task trees in TCA). However, they differ in three aspects. First, TCA allows the exception handlers to manipulate the task trees explicitly, while explicit manipulation of the call-stack in Ada is prohibited. Second, popping and pushing the call-stack is always simpler than killing and adding new subtrees, because of the temporal constraints placed on the task trees. Maintaining the desired temporal constraints between tree nodes while modifying the task trees is a difficult problem, which we have not solved completely. Third, task tree nodes are not killed while TCA is searching for capable handlers, so the exception handlers can examine the failed node and its ancestors to help in debugging [14].

7. Conclusion

We have designed and implemented TCA, a general-purpose task control architecture, for the control of mobile robots. TCA is designed to be used for robots with multiple tasks, and limited computational and physical resources, that operate in an uncertain and changing, but relatively benign, environment. The design of TCA is based partly on experience gained from our first version of the Hero testbed. That version, developed in an ad hoc manner, had several shortcomings, such as brittleness, unawareness of environmental changes, etc. By using TCA, we have re-implemented the system in a more disciplined way. The current robot can navigate in a changing (indoor) environment, avoid obstacles, collect cups on the floor, and at the same time watch for failures and contingencies, recover from failures, and go recharge when necessary.

The features of TCA that result in the Hero robot's success and that, we believe, will facilitate the building of intelligent, robust robots are (1) distributed processing, (2) resources, (3) task trees and temporal constraints, (4) concurrent monitors, and (5) context-dependent exception handling. The distributed environment enables robot activities such as planning, sensory data processing, monitoring, and plan execution, to be performed concurrently. The resource mechanisms enable robots to schedule the use of their limited resources. By using the temporal mechanisms, the user can implement intelligent robots that are able to act on an incomplete plan when not enough information is available to make a decision, and to take advantage of parallelism by planning ahead when needed information is obtainable. Concurrent monitors, which allow robots to acquire information from the environment while executing tasks, gives robots the opportunity of reacting to environmental changes and changing their focus for contingencies or opportunities. The exception handling mechanisms enable robots to dynamically choose context-dependent strategies for handling contingencies, planning time failures, and execution time errors. The mechanisms also allow robots to re-use a failed plan by making changes in it, or even to change their focus completely.

Another important feature of TCA is that it facilitates modular and incremental design of complex robot systems. In TCA, planning, execution, monitoring, and exception handling are all logically and functionally separate activities. This enables one to build systems incrementally -- first building behaviors that plan and execute, then adding features (usually by adding new code with few changes to the existing programs) to take advantage of concurrency in planning and execution, to monitor for exceptional situations, and to handle those situations intelligently.

Despite these encouraging results, much more work remains to be done. In particular, we plan to extend TCA to support various knowledge-intensive decision-making capabilities [8], such as, scheduling various tasks based on their urgency and relative cost, choosing optimal plans based on the analysis of

various plans' strength, limitation, resource usages, time constraints, etc.

Although building complex, robust robot systems is still very much an art, we believe that with the use of high-level architectures, such as TCA, we can make the process easier. Through experience with different robot systems (the CMU planetary Rover also uses TCA), and analysis of the requirements for different environments and robot configurations, we are converging on a set of mechanisms to support the building of such robot systems.

8. Acknowledgements

We express our gratitude to Robert Eric Wolpov, who has been adding new sensors to the Hero robots and maintaining them in good health. We thank Tom Mitchell and Andrew Phillips, who helped develop the first version of the testbed. Lonnie Chrisman has provided valuable comments on TCA design and Kevin Ryan helped develop the guarded commands. We are grateful to them. We also thank John Allen, Jim Moody, and Steve Shafer for their assistance in setting up the testbed. This research has been supported by NASA under Contract NAGW-1175.

References

[1] Agre, P.E., Chapman, D.
Pengi: An Implementation of a Theory of Activity.
In *Proceedings of AAAI-87*, pages 268-272. 1987.

[2] Amould, E.A., Bitz, F.J., Cooper, E.C., Kung, H.T., Sansom, R.D., and Steenkiste, P.A.
The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers.
Technical Report, CMU-CS-89-101, Carnegie Mellon University, 1989.

[3] Bares, J., et al.
Ambler: An Autonomous Rover for Planetary Exploration.
In *IEEE Computer, Vol. 22, No. 6.* 1989.

[4] Brooks, R.A.
A Robust Layered Control System for a Mobile Robot.
In *IEEE Journal of Robots and Automation, vol. RA-2, no. 1.* 1986.

[5] Firby, R.J.
Adaptive Execution in Complex Dynamic Worlds.
Technical Report, YALEU/CSD/RR #672, Yale University, 1989.

[6] Flynn, A.M., and Brooks, R.A.
MIT Mobile Robots - What's Next?
In *Proceedings IEEE Robotics and Automation*, pages 611-617. April, 1986.

[7] Georgeff, M.P.
A System for Reasoning in Dynamic Domains: Fault Diagnosis on the Space Shuttle.
Tech Note 475, AI Center, SRI International, 1986.

[8] Georgeff, M.P., Ingrand, F.F.
Decision-Making in an Embedded Reasoning System.
In *Proceedings of IJCAI-89*, pages 972-978. 1989.

[9] Habermann, A.N., and Perry, D.E.
Ada for Experienced Programmers.
Addison-Wesley Publishing Company, Inc., 1983.

[10] Hamey, L., Printz, H., Reece, D., and Shafer, S.A.
A Programmer's Guide to the Generalized Image Library
Carnegie Mellon University, 1987.

[11] Kaelbling, L.P.
An Architecture for Intelligent Reactive Systems.
Tech Note 400, AI Center, SRI International, 1986.

[12] Lin, L.J., Mitchell, T.M., Phillips, A., and Simmons, R.
A Case Study in Autonomous Robot Behavior.
Technical Report, CMU-RI-89-1, Robotics Institute, Carnegie Mellon University, 1989.

[13] Simmons, R.
Commonsense Arithmetic Reasoning.
In *Proceedings of AAAI-86*, pages 118-124. 1986.

[14] Simmons, R.
A Theory of Debugging Plans and Interpretations.
In *Proceedings of AAAI-88.* 1988.

[15] Simmons, R., Mitchell, T.M.
A Task Control Architecture for Mobile Robots.
In *Stanford Spring Symposium*. 1989.

[16] Smith, R.C., and Cheeseman, P.
On the Representation and Estimation of Spatial Uncertainty.
In *The International Journal of Robotics Research*, pages 56-68. 1986.