Structured Control for Autonomous Robots

Reid G. Simmons

Abstract— **To operate in rich, dynamic environments, autonomous robots must be able to effectively utilize and coordinate their limited physical and computational resources. As complexity increases, it becomes necessary to impose explicit constraints on the control of planning, perception, and action to ensure that unwanted interactions between behaviors do not occur.**

This paper advocates developing complex robot systems by layering reactive behaviors onto deliberative components. In this *structured control* approach, the deliberative components handle normal situations and the reactive behaviors, which are explicitly constrained as to when and how they are activated, handle exceptional situations.

The *Task Control Architecture* (TCA) has been developed to support this approach. TCA provides an integrated set of control constructs useful for implementing deliberative and reactive behaviors. The control constructs facilitate modular and evolutionary system development: they are used to integrate and coordinate planning, perception, and execution, and to incrementally improve the efficiency and robustness of the robot systems. To date, TCA has been used in implementing a half-dozen mobile robot systems, including an autonomous six-legged rover and indoor mobile manipulator.

I. Introduction

TO CARRY OUT complex tasks in rich, dynamic environments, autonomous robots must decide when to plan and when to act, how to detect and recover from errors, how to handle conflicting goals, etc. In short, the robots must effectively coordinate their limited physical and computational resources. As the tasks and environments become increasingly complex, explicit constraints that impose structure on the control of planning, perception and action are needed to improve system understandability and to ensure that the robots will achieve their tasks.

A current methodology is to design robot systems as collections of *behaviors* that are independent, action-generating entities. In the *reactive* approach, systems consist of collections of local behaviors that are triggered by direct sensing of the environment [6], [10], [17]. The global, goal-directed behaviors of such systems are not explicitly planned: they typically emerge from interactions between the local entities [1], [7]. While this approach handles uncertainty and unpredictable changes well, it is unclear how it scales as tasks and environments increase in diversity.

The problem is that as complexity increases, interactions between behaviors increase as well, to the point where it becomes difficult to predict the system's overall behavior. One way to limit interactions is to add top-down constraints that take advantage of regularities in the domain to coordinate actions. This strategy is embodied in the *deliberative* approach, which hierarchically partitions problems into manageable subtasks and explicitly controls interactions between them.

A limitation of the deliberative approach is that strict top-down constraints may prevent the system from being responsive to changes in the environment. Thus, a combination of deliberative ("feedforward") and reactive ("feedback") behavior is needed to deal with complex, dynamic domains [20]. The approach advocated here, which we term *structured control*, is to start with basic deliberative components that handle nominal situations. System reliability is increased by incrementally layering on reactive behaviors to handle exceptions. Explicit temporal and resource constraints delimit the contexts in which the behaviors are active.

The structured control approach of layering constrained, reactive behaviors onto a deliberative base provides an engineering basis for designing autonomous robot systems. First, the separation of nominal and exceptional behaviors increases system understandability by isolating different concerns: the robot's behavior during normal operation is readily apparent, and strategies for handling exceptions can be developed as needed. Furthermore, complex interactions are minimized by constraining the applicability of behaviors to specific situations, so that only manageable, predictable subsets will be active at any one time. Finally, the structured control approach acknowledges that creating complex robotic systems is an incremental process: designers should be able to add new behaviors with little or no modification to existing systems.

The *Task Control Architecture* (TCA) has been developed as a framework for combining deliberative and reactive behaviors to control autonomous robots. The term *task control* refers to the problem of coordinating perceptual, planning, and execution components of a robot system to achieve a given set of goals. Task control problems include noticing that goals need attention, deciding which goals to attend to, constructing plans (to some level of detail), monitoring progress, and dealing with exceptions.

While the Task Control Architecture does not itself provide behaviors for particular tasks, it does provide designers with control constructs for developing such behaviors and software utilities for implementing the necessary control decisions. In essence, TCA is a high-level robot operating system that provides an integrated set of commonly needed mechanisms to support distributed communications, task decomposition, resource management, execution monitoring, and error recovery.

A robot system built using TCA consists of task-specific modules that communicate by sending messages via a general-purpose, reusable *central control module*. The task modules perform all robot-dependent information processing, while the

Manuscript received March 19, 1992; revised December 17, 1992. This research is supported by NASA under contract NAGW-1175.

The author is with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

IEEE Log Number 9215093.

central control module is responsible for routing messages and maintaining task control information. The task modules use the TCA control constructs to specify information such as how to decompose tasks, when subtasks should be planned and executed, when and how to monitor the environment, and how to react to exceptional situations (by terminating tasks, adding new subtasks, reordering tasks, etc). TCA utilizes this control information, in turn, to schedule and coordinate the actions of the modules and to respond to change in a context-sensitive fashion.

Most of the TCA control constructs can be incrementally added with little or no change to existing systems. In particular, modules can be added that perform new tasks, temporal constraints can be modified to increase concurrency, new monitors and exception handlers can be easily added to existing tasks, and new implementations can replace existing modules without changing communication protocols. We have found that these capabilities greatly facilitate the development of complex systems.

To date, the Task Control Architecture has been used in over a half-dozen mobile robot systems, including a six-legged robot [27], an indoor mobile manipulator [29], a coal-mining robot [23], a system to inspect the tiles of the Space Shuttle [11], and a NASA robot for space station assembly [14]. The first two systems will be used as the main examples in this paper.

The Ambler (Fig. 1) is a six-legged robot, designed for planetary exploration, that autonomously traverses rugged terrain [27]. The Ambler system uses TCA to integrate real-time control, 3D perception, planning algorithms, monitoring, and error recovery procedures. The deliberative aspects of TCA are used to plan safe and energy efficient moves, based on a host of kinematic, pragmatic, and terrain constraints [28]. The reactive components are used to detect and handle deviations arising from sensor and actuator uncertainty [25].

The indoor mobile manipulator is based on a Hero 2000, a wheeled robot with an arm and two-fingered gripper. A ceiling-mounted camera provides a "satellite" view of our laboratory (Fig. 2). The Hero operates in a peopled, unmodified office environment. Its tasks include collecting cups off the floor, retrieving printer output, delivering objects to workstations, avoiding static and dynamic obstacles, and recharging itself when necessary [15], [29].

The next section discusses the issues of deliberation and reactivity, and how other robot architectures address these issues. Section III describes the Task Control Architecture in some detail, focusing on how it supports the design of deliberative and reactive behaviors. Section IV describes how TCA facilitates incremental system development, and Section V presents conclusions.

Fig. 1. The six-legged Ambler robot.

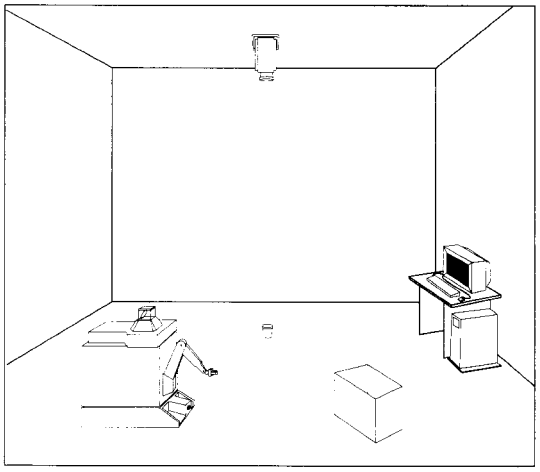

Fig. 2. The Hero mobile manipulator.

II. Deliberation and Reactivity

We take the essence of the deliberative approach to be the top-down decomposition of tasks into subtasks, specifying current and future activities and constraints (temporal, resource) among them. Reactivity means that the system detects and makes appropriate responses to changes in its environment. In short, deliberation is more concerned with the way subtasks interact, and reactivity is more concerned with the "here and now." Note that "reactive" does not necessarily imply "reflexive" (direct connections between sensors and effectors)—any computation is allowable as long as the system is "fast enough" to respond to the given situation.

Both approaches have strengths and weaknesses as frameworks for robot architectures. The deliberative approach can handle complex tasks by breaking them into more manageable subproblems. Planning and search can help the robot avoid local traps in the environment, which can decrease risk and increase reliability. It is more difficult for reactive architectures to deal with behavioral interactions, unless they are known and specified in advance. On the other hand, the task decom-

position is only as good as the models and information the robot has. Often, decisions must be deferred until the robot has access to the required information.

The reactive approach provides for robot safety by enabling the system to be responsive to the environment. This can be invaluable if the environment is uncertain or unpredictable. Also, response time can often be guaranteed for reactive architectures—this is more difficult for deliberative architectures, particularly if the depth of the task decomposition can be unbounded.

In many cases, different architectures can be used for the same tasks. For example, both reactive and deliberative architectures have been used for retrieving objects [10], [29] and for controlling walking robots [6], [27], albeit using very different control structures. Typically, the differences between architectures are the ease with which systems can be developed and the efficiency with which tasks can be achieved. We contend that for robots with complex tasks in rich environments, the structured control approach simplifies development because behaviors can be separated and interactions constrained. This can lead to more predictable and maintainable systems.

Several primarily deliberative architectures have been proposed. The work of Albus [2] and Meystel [18] promote the idea of top-down, hierarchical controllers, each executing a sense-plan-act feedback loop. The NASREM architecture [3], in particular, is a strict hierarchical framework for task decomposition, perception and world modeling. The hierarchy is based on temporal abstraction—each level deals with events that characteristically occur an order of magnitude slower than those of the level below. With TCA, in contrast, task decomposition is based on the functionality needed and the degree of interaction between subtasks: in TCA-based systems, the width and depth of the decomposition depends largely on task complexity. In addition, TCA provides more guidance for determining how to design and combine deliberative and reactive behaviors.

The BB1 blackboard architecture [16] provides many of the same advantages as TCA in terms of decomposing, coordinating and scheduling subtasks. Its use of a *control plan* facilitates placing temporal and resource constraints on subtasks, setting up monitors, and controlling the interleaving of planning and execution. The centralized blackboard and "whiteboard" [22] representations of such architectures, however, are potential performance bottlenecks [7], [10]. While TCA does maintain control information centrally, the actual data needed to solve problems is distributed amongst the system's processes. This separation makes sense because control decisions are made relatively infrequently, while centralized data would be more of a bottleneck due to the higher bandwidth of communication required.

Primarily reactive architectures are exemplified by Brooks' Subsumption architecture [5], [10] and other behavior-based approaches [4], [17], [21]. In these architectures, the overall system behavior emerges from the interactions of local, often reflexive, behaviors. The architectures differ primarily in how behaviors are combined and interact. In the Subsumption architecture, for example, behaviors can directly inhibit the actions of others. While such a fixed priority scheme is simple, it is often difficult to determine how to prioritize competing behaviors. Most other behavior-based architectures use a more general arbitration scheme, in which the recommendations of all competing behaviors are taken into account—either by choosing a single action [17] or combining them [4].

The difficulty with such approaches is that as the number of behaviors and their degree of interaction grow, it becomes increasingly difficult to design good arbitration schemes or even to understand how the system will behave in general. One approach is to add more structure to the reactive behaviors. For instance, a deliberative component can be used to partition behaviors into subsets that are activated in certain situations [13], [21]. While this is similar to the structured control approach we are advocating, TCA provides control constructs that not only structure the interactions between behavior sets, but within them as well.

The RAPs (Reactive Action Packages) architecture [12] is probably most similar to TCA in concept. In RAPs, task decomposition, monitors, and error recovery procedures are packaged into discrete units. A RAP defines methods for decomposing tasks into subtasks, how to detect the successful achievement of the task, and methods for responding to changes. The main differences are that RAPs is more concerned with reactive behaviors and real-time response, while TCA is more concerned with structuring interactions to handle complex tasks. In addition, reactive behaviors are more fully integrated into RAPs, while TCA cleanly separates the various aspects of robot control, enabling plan formation and execution to be developed independently of monitoring and error recovery. In general, on the reactive/deliberative spectrum, systems using TCA tend to be more deliberative than those using RAPs.

III. The Task Control Architecture

The Task Control Architecture provides a comprehensive set of control constructs for developing deliberative and reactive robot behaviors. The constructs are designed to integrate smoothly in order to provide predictable aggregate behavior. The control constructs include support for:

1) Distributed inter-process communication
2) Task decomposition and temporal constraints between subtasks
3) Resource allocation and management
4) Execution monitoring
5) Exception handling

At the basic level, TCA supports distributed processing and communication. A robot system utilizing TCA consists of a number of robot-specific modules (C and/or Lisp processes), and a *central control* module, which is common to all systems that use TCA (Fig. 3). Modules communicate by passing coarse-grained messages to the central control, which routes them to be handled by the appropriate modules. Message routing information is determined dynamically when modules connect with the central control: modules register message names, formats of the data structures associated with the messages, and message handling procedures.

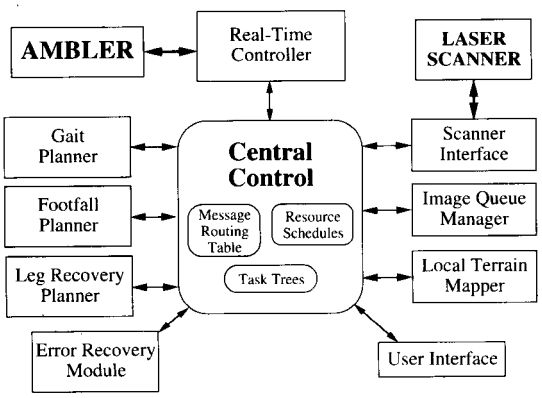

Fig. 3. Modules for the Ambler walking system.

Modules send messages by name. This facilitate the use of "plug compatible" modules, since the sender need not know which modules will handle the messages. For example, the Ambler planning modules are developed using a graphics simulator that has the same interface as the real-time controller. Once developed, the planners can run the actual robot without any change.

An expressive data description language enables TCA to pass fairly complex data, including structures, pointers, arrays, strings, etc. TCA automatically converts the data into a byte stream, passes it via sockets over the Ethernet to the central control module, determines which module has registered a handler for the message, forwards the message to the module, reassembles the data, and invokes the appropriate message handling procedure. Even though messages are centrally routed, message passing is fairly efficient since the data is not actually interpreted by the central control module. For messages up to several kilobytes, the round-trip message passing time is typically less than 100 ms.

TCA provides several classes of messages, each with somewhat different semantics:

1) *Inform* messages are one-way communications, used to asynchronously pass information from one module to another.
2) *Query* messages are two-way, returning a reply to the sending module. Query messages are blocking, pending receipt of the reply.
3) *Goal* messages are used to decompose tasks into subtasks. They are non-blocking, to enable planning and plan execution to occur concurrently.
4) *Command* messages are similar to goal messages, but they represent executable actions of the robot system.
5) *Monitor* messages are used to set up execution monitors.
6) *Exception* messages are used for handling exceptional situations.

The effects of the last four message classes will be described in more detail in subsequent sections. In the next section, we present an example of how TCA is used in the control of an autonomous mobile robot.

A. An Example

A TCA-based system consists of the central control module and a number of modules, running on one or more computers, written specifically for the particular robot. To run the system, one invokes the central control process, indicating how many modules will be running and where to log the message traffic, if desired. Each robot-specific module is then started. The modules communicate with the central control using a library of function calls [26]. Modules first connect to the central control module, and register which messages they handle, their formats, and other relevant information, such as the monitors and exception handlers used by the system. Modules then call a function that waits to receive messages and dispatches them to the appropriate handlers.

For the Ambler system (Fig. 3), walking is initiated by entering a series of waypoints at a graphical user interface. When a route is determined, the user interface module sends a series of "Traverse Arc" goal messages, one for each waypoint (Fig. 4). The central control forwards the first "Traverse Arc" message to the gait planner module, queuing the remaining ones. Within the gait planner, TCA decodes the message data and invokes the appropriate message handling procedure. This procedure issues a query message, requesting the Ambler's current position. TCA forwards the request to the controller module, and routes the response back to the gait planner. The gait planner calculates an arc that passes from the current location to the goal location, and issues a "Take Steps" goal message, passing the current position and desired arc to follow. When the message handler completes, the central control is notified that the gait planner is no longer busy.

When the gait planner is free, TCA forwards it the "Take Steps" message, invoking the appropriate procedure. This handler first checks whether the robot is at the goal location. If so, it returns immediately; if not, it calculates a body move that maximizes forward progress. It then issues a query to determine the best place to move the leg. The footfall planner handles this message by querying for an elevation map (handled by the local terrain mapper) and computing the desired information. Once the calculations are completed, the gait planner issues a "Place Leg" goal message, a "Move Body" command, a monitor to ensure that the move was executed correctly, and another "Take Steps" message (Fig. 4). TCA immediately forwards the "Place Leg" message to the leg recovery planner, which obtains an elevation map and uses it to compute an energy-efficient path for the leg to follow. The leg recovery planner then issues a "Move Leg" command, which is routed to the controller. Concurrently, TCA sends the pending "Take Steps" message to the gait planner, which begins to plan out the next step.

After the controller finishes handling the "Move Leg" command, it is forwarded the "Move Body" command. When the body move completes, the central control activates the monitor, sending a query to see if the desired move was achieved. If not, an exception is raised and the gait planner is invoked to replan from the current position. Otherwise, TCA waits for the next "Move Leg" message to be issued, and forwards that to the controller, beginning another walking cycle.

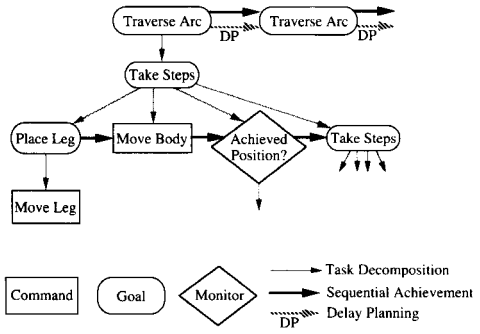

Fig. 4. Task tree for autonomous walking.

The Ambler modules instruct TCA to perform several other tasks during walking. The scanner interface module indicates that it should be notified after each "Move Body" message completes, in order to acquire a new laser range image. The error recovery module instructs TCA to invoke a stability monitor prior to executing each leg and body move; if the planned move is found to be potentially unstable, the monitor aborts the walking task and the system comes to a graceful halt. Additional mechanisms used to increase the reliability of the Ambler system are described in Section III–C.

B. Mechanisms for Deliberation

At the heart of TCA is a hierarchical representation of task/subtask relationships. This representation, called a *task tree*, has goal messages as non-terminal nodes, and executable command and monitor messages at the leaves (Fig. 4). TCA constructs task trees automatically: whenever a goal, command, or monitor message is issued, the central control creates a node and adds it as a child of the node that issued the message. Temporal constraints between nodes (illustrated using heavy horizontal arrows) are used to schedule task planning and execution: messages are queued until their temporal constraints are satisfied. This combination of hierarchical task decomposition and temporal constraints form TCA's representation of plans.

As Fig. 4 illustrates, TCA task trees may be arbitrarily deep and subtrees may be expanded to different depths. This contrasts with other architectures, such as NASREM [3], in which the hierarchical structure is fixed. Our experience with the Ambler and Hero robots indicates that this increased flexibility far outweighs the run-time overhead of maintaining the task tree representation.

When modules send messages, they may include temporal constraints to inform TCA when to dispatch the messages. The temporal constraints indicate relationships between subtasks within a task tree. A *sequential-achievement* constraint between two nodes implies that all command and monitor messages under the first node (the leaves of its subtree) must be handled before any of those under the second node. For example, the sequential-achievement constraint between the "Traverse Arc" nodes in Fig. 4 indicates that the arcs must be traversed in order. Partially ordered plans are denoted by the absence of sequential-achievement constraints between nodes.

A *delay-planning* constraint implies that a goal message should not be handled (decomposed into subgoals) until the previous task has been completely achieved. The delay-planning constraint between the "Traverse Arc" nodes indicate that the system should not begin planning how to follow one arc until the previous arc has been negotiated. On the other hand, the lack of an explicit delay-planning constraint between the "Achieve Position?" monitor and the subsequent "Take Steps" goal indicates to TCA that the gait planner can concurrently plan one step while the previous leg and body moves are being executed (although the sequential-achievement constraint between them indicates that the steps must still be *executed* in order).

The temporal constraints are implemented by associating several time intervals with each task tree node. A *handling interval* denotes the period during which the node's message is actually processed by a module. An *achievement interval* denotes the execution time of a task: the time taken by all command and monitor messages in the subtree. For goal nodes, an additional *planning interval* is defined as the time taken to handle all the goal nodes in the tree; intuitively, this represents the time needed to completely expand the subtree to its executable primitives.

Temporal constraints are expressed in terms of the start and end points of these intervals (a formalization is found in [24]). For example, the delay-planning constraint between nodes $n1$ and $n2$ is defined as: $Achievement_{n1}.end \leq Planning_{n2}.start$. TCA dispatches a message only when all temporally preceding messages have been handled. Temporal precedence holds between subtasks and supertasks, so that the end of the achievement (and planning) of subtasks precedes the end of their parent task. For example, if the "Move Body" command in Fig. 4 is being handled, TCA can infer that the second "Traverse Arc" goal is not yet ready to be dispatched.

Another deliberative method for controlling tasks is to explicitly manage the robot's physical and computational resources. A TCA *resource* is a set of message handling procedures and a capacity. TCA ensures that a resource's capacity is never exceeded, queuing messages if necessary until the resource becomes available. By default, TCA groups all message handlers registered by a module into a resource of unit capacity. For example, the Ambler's gait planner handles the "Traverse Arc" and "Take Steps" messages, among others. TCA ensures that the module will receive only one of these messages at a time. Modules may also define additional resources, enabling them to handle multiple messages concurrently. The Hero controller module, for example, defines one resource for its actuator commands and one for its sensor queries, enabling the robot's position to be tracked while it is moving.

For finer control, a module can *lock* a resource, preventing other modules from accessing the resource until it is unlocked. While this raises the possibility of deadlock, judicious use of resource locking can increase overall resource utilization and system predictability. The Hero system, for instance,

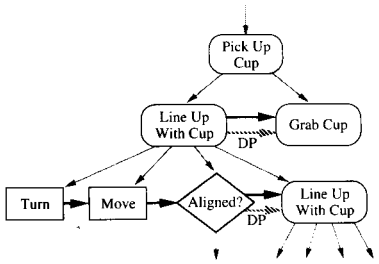

Fig. 5. Feedback loop for aligning with an object.

uses resource locking to enable the planners to gain high-priority access to the overhead camera. To prevent blurring, the Ambler's scanner interface module locks the resource of the controller module to preclude movement during image acquisition.

Using the task tree, temporal constraint, and resource mechanisms, complex robot systems can be developed that integrate concurrent, distributed planning, perception and real-time control. The mechanisms enable behaviors to be scheduled so as not to interfere with one another. The next section describes how reactive behaviors can be added to make systems more reliable.

C. Mechanisms for Reactivity

To operate in uncertain, dynamic environments, autonomous robots must be reactive to change. TCA provides several constructs for monitoring changes in the environment. A TCA *monitor* is a message that performs some action when a specified condition is triggered. A *point monitor*, which tests its condition just once, is useful for determining whether tasks have been executed according to plan. For example, the point monitor in Fig. 4 compares the actual Ambler position against its planned position. If the difference exceeds a given threshold (e.g., if the robot slipped), the monitor's action is triggered to replan subsequent steps.

Point monitors can be combined with TCA's task tree and temporal constraint mechanisms to create high-level feedback loops: to achieve a goal, the robot performs some action, monitors to see if the goal is achieved and, if not, recursively tries to achieve the goal. In picking up a cup, for instance, the Hero robot first aligns itself with the cup. This involves turning and moving (Fig. 5), followed by monitoring the results to determine whether the motion was accurate enough for grasping to succeed. If not, the monitor simply terminates, which enables the succeeding "Line Up" goal to be handled, recursively planning out the next sequence of moves. If the robot position is acceptable, the monitor's action message deletes the next "Line Up" goal from the task tree, effectively terminating the servo loop.

Polling and *interrupt-driven* monitors, which are used to detect unexpected changes, operate concurrently with planned actions. Both types of monitors test their conditions repeatedly, continuing either for a given duration or until a specified event occurs. For polling monitors, the central control module issues the condition message at a fixed frequency. For interrupt-driven monitors, TCA informs modules when to set up new monitors and when to cancel them. The modules have responsibility for informing the central control whenever the monitor's condition holds, at which point TCA issues the associated action message. Polling and interrupt-driven monitors provide different run-time and design-time advantages: while interrupt-driven monitors are often more efficient in their demands on perception, it is typically easier to implement polling monitors since TCA handles their activation and scheduling.

An important aspect of the structured control approach is that the context in which reactive behaviors are applicable should be constrained to minimize both unexpected interactions and the load on perception [8]. For example, the Hero system uses a polling monitor to check its battery once a minute, inserting a "recharge" task if the level is low. The monitor is activated only when the Hero is off its charger, and it is deactivated as soon as the monitored condition is met. Similarly, when a cup-collection task is initiated, an interrupt-driven monitor is activated that lasts until the cup is grasped. Whenever an overhead image is acquired, the monitor checks whether the cup is still visible, cancelling the collection task if the object disappears from view (e.g., someone else picked it up).

In TCA, monitors can be added without modifying existing deliberative components. A module can associate a monitor with a class of messages and constrain the monitor to be triggered either before, during, or after the associated message is handled. This is called the "wiretap" mechanism because whenever a message of the specified class is handled, by whichever module, TCA invokes the monitor, sending it a copy of the message data. For example, before every Ambler move a stability monitor is invoked to verify that the move will not cause the robot to tip over, and after every leg move a footfall monitor analyzes the force sensor data to detect possibly unstable footholds (Fig. 6).

Monitors are one way that exceptional conditions can be detected through TCA. Message handlers can also detect execution errors or plan failures directly, which they signal by raising exceptions. For example, a controller module may sense that a motor is stuck, or a planning module may not be able to find a clear path to a goal. *Exception* messages can be associated with task tree nodes to deal with such conditions. Exception handling is structured hierarchically: when an exception is raised, TCA searches up the task tree to find and dispatch a message designated to handle that exception. If the message handler determines it cannot actually fix the problem, it reissues the exception and the search continues up the tree. If the root node is reached, TCA simply terminates the task.

The motivation behind invoking lower-level exception handlers first is the expectation that they will produce specific, minimal patches to the plan. To illustrate, Fig. 6 shows various exception handlers layered onto the nominal Ambler walking plan. If a leg or body move command fails, the first strategy employed is to just retry the move. If this fails, the system tries replanning the move, since the Ambler

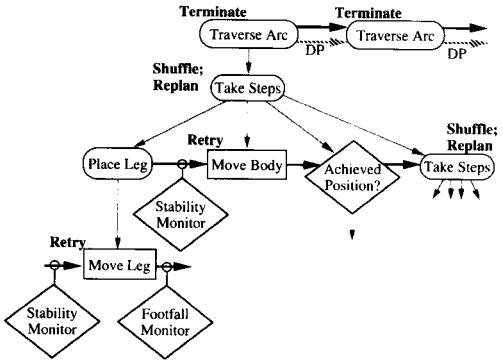

Fig. 6. Ambler task tree with monitors and exception handlers.

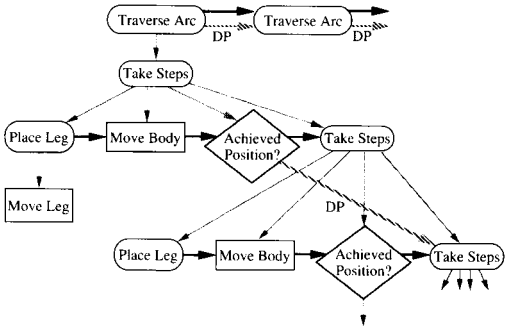

Fig. 7. Concurrent planning and execution.

configuration might have changed between the planning and execution of the move. Failing this, the next strategy is to shuffle the legs, moving them into a standard stance, and then to replan from the new configuration. If the legs cannot be shuffled (e.g., moving legs would cause the Ambler to become unstable), the walking task is terminated altogether.

Whichever way an exceptional situation is detected, the response often involves altering the plan being executed. TCA provides facilities that enable modules to examine and modify task trees. Examination facilities include finding parent, children, and sibling nodes, querying for the status of a node, and retrieving the data associated with messages in the task tree. Task tree modifications include resending messages, terminating tasks (and their subtrees), inserting new nodes, and adding additional temporal constraints. These mechanisms provide fairly general capabilities for repairing and patching faulty plans.

The monitoring and exception handling facilities have been used to make the Ambler system reliable enough to autonomously walk hundreds of meters in rough terrain. In one set of experiments [25], the Ambler took over 700 steps, covering some 300 meters in boulder-strewn, sandy terrain. Exceptional situations were detected in about 18% of the steps, and the system recovered autonomously in all but two situations. More recently, the Ambler walked outdoors 500 meters contiguously. The terrain was somewhat more benign—hilly, but not rock-strewn—and the exceptional, reactive behaviors were needed in only about 8% of the 1200 moves.

For some errors, the message-passing delay of TCA (50–100 ms) provides insufficient response time. In such cases, reflexive behaviors are used that quickly stabilize the robot and then invoke the hierarchical exception-handling mechanism of TCA to provide a reasoned response to the problem. For example, the Ambler's controller module monitors force sensors in the feet to detect terrain contact, halting the robot's motion (within 5 ms) and signaling TCA when an unexpected collision occurs. While these reflexive behaviors are currently implemented outside of TCA, we are investigating ways to integrate them into the framework.

IV. Incremental Development of Robot Systems

Design of any complex system is an iterative process. This is especially true for robot systems, since the nature of their environment (including real-time requirements) may not be well understood from the start. The structured control approach facilitates this by enabling new behaviors, both deliberative and reactive, to be added incrementally, with minimal impact to the existing software system.

In particular, monitors and exception handlers can be added without modifying existing components. The wiretap mechanism is used to associate monitors with specific messages, so that the monitor is activated whenever the associated message is handled. Similarly, a module can associate an exception handler with a class of messages handled by another module, so that whenever a message is issued and added to the task tree, the exception handler is automatically added to that node.

One difficulty is to prevent harmful interactions between behaviors that are developed independently. A useful methodology is to initially overconstrain the system, then slowly relax constraints until the desired performance is obtained. The Ambler system, for instance, was originally developed with a sequential sense-plan-act cycle. After sufficiently testing the sequential version, temporal constraints were removed, enabling one step to be planned concurrently with the achievement of the previous step [24].

This modification, by itself, led to problems: since the planning algorithms are fast relative to the robot's motions, we found that the planner soon got several steps ahead, operating near the Ambler's perceptual horizon where the data is very uncertain. To prevent this, a delay-planning constraint is added between one "Take Steps" node and the node directly preceding its parent (Fig. 7). This yields one step lookahead planning; for two step lookahead, we would use the grandparent node, etc.

The addition of concurrency increased average walking speed by over 30%, with only minor modifications to the existing system [24]. Fig. 8 presents a graph, produced by a program that analyzes TCA log files, of module utilization for the concurrent system. The darkly shaded areas indicate when modules are computing; lightly shaded areas are when they are awaiting replies from other modules. As can be seen, this produces nearly continuous motion of the machine: the

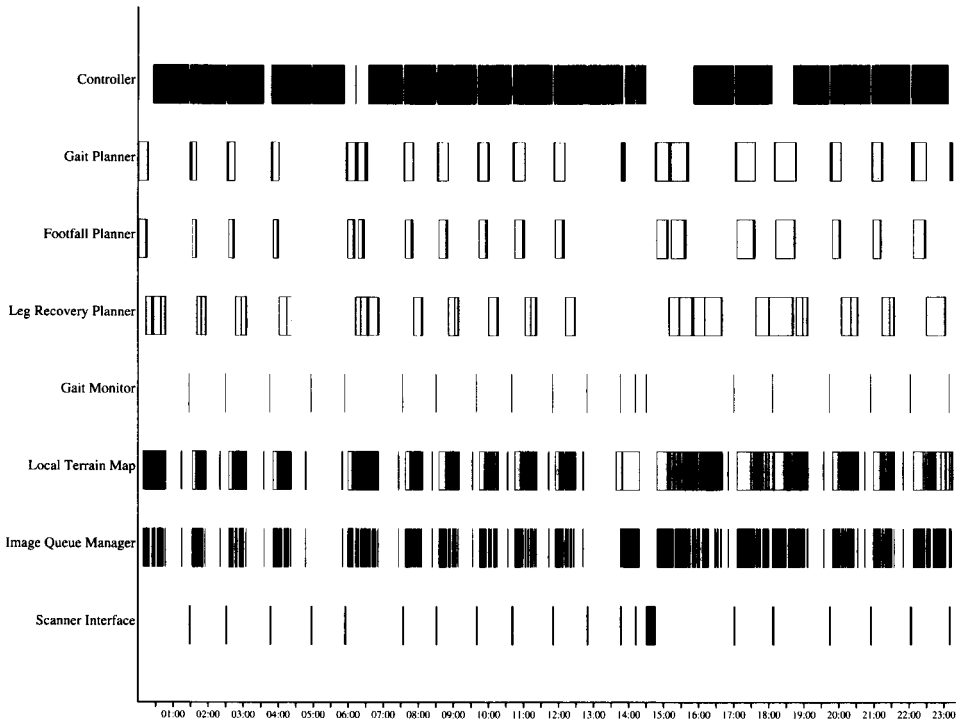

Fig. 8. Module utilization for Ambler walking.

controller module is active about 80% of the time, while planning and perception take about half that time to produce suitable gaits.

Often, decisions about how far to plan in advance, and to what level of detail, depend on the uncertainty in the robot's current knowledge and its expectations about the future. To the extent that the world is predictable and stable, advance planning is warranted. In unpredictable domains, however, more reflexive behavior is appropriate. While TCA itself does not address these knowledge-level issues, its task trees and temporal constraint mechanisms provide the necessary framework for expressing and acting upon such decisions. In effect, TCA treats planning as a computational activity that must be scheduled along with other actions, such as robot motion and sensing. By suitably constraining the achievement and planning intervals of task tree nodes, modules can implement a wide variety of strategies for interleaving planning and execution [19].

As an example, Fig. 9 presents a simplified version of the task tree created by the Hero system for collecting a cup. It turns out that the Hero cannot plan in advance how to pick up the cup because it does not know what kind of cup it will encounter until it gets close enough. This is expressed using a delay-planning constraint between the "Navigate to Cup" and "Pick Up" nodes. On the other hand, when the Hero begins

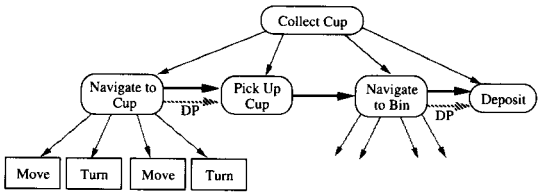

Fig. 9. Task tree for Hero cup collection.

the cup-collection task it has enough information (using its overhead camera) to plan how to get from the cup to the trashbin. The decision that advance planning can occur here is indicated by the absence of any planning constraints between the "Navigate to Bin" and other nodes. Thus, the system can plan the path while moving to the cup; once it has grasped the cup it can immediately begin executing the planned path, without having to wait for the time-consuming path planning process.

Since perception algorithms are typically computationally expensive, use of concurrent perception can markedly improve response time. The Ambler system was incrementally modified from acquiring images on demand to acquiring them asynchronously. In particular, the wiretap mechanism is used to

notify the perception subsystem to acquire a laser range image whenever the Ambler has moved. A similar modification to the Hero—asynchronously processing overhead images—nearly halved the average time needed to accomplish cup-collection tasks [29].

V. Conclusions

The Task Control Architecture provides a framework for developing and controlling autonomous robot systems. The underlying philosophy is that *the control of planning, perception, and action must be well-structured for general-purpose robots to succeed in rich and uncertain environments*. While reflexive control strategies may suffice for simple tasks, robots with complex tasks and environments need to effectively manage their limited resources and intelligently coordinate their actions to eliminate unwanted interactions.

To facilitate development of both deliberative and reactive behaviors, the Task Control Architecture provides common control constructs, including distributed communications, hierarchical task decomposition, temporal constraints to coordinate subtasks, resource management, monitoring, and exception handling. The constructs are designed to support the *structured control* approach, in which deliberative components that handle nominal situations are layered with reactive behaviors, constrained to limit their potential for unwanted interaction. TCA and the structured control methodology have been used in over a half-dozen robot systems, including a six-legged robot that autonomously walks over rugged terrain and an indoor mobile manipulator operating in a peopled environment.

It is clear from experience with TCA that using formal methods to analyze the constraints on behavior would greatly facilitate development of robot systems. We are moving in that direction, starting by formally defining the various control constructs [24]. We intend to utilize temporal verification systems [9] to ensure that the constraints imposed are sufficient to meet the robot's functional specifications. Preliminary results indicate that the types of control constructs provided by TCA are directly amenable to such analysis.

We contend that the use of structured control facilitates the development of complex robots. The explicit use of constraints provides a basis for precisely characterizing, designing, and analyzing interactions between behaviors. Implementation is facilitated by providing mechanisms that map directly from design decisions to methods of communication, task coordination, and reactivity. Finally, systems can be developed incrementally by layering on new behaviors and new constraints. These contribute to producing autonomous robot systems that are competent, reliable, and understandable.

VI. Acknowledgement

Christopher Fedor, Richard Goodwin, and Long-Ji Lin have contributed much to the design and implementation of TCA. Many members of the Hero and Ambler Planetary Rover projects have used and helped refine the architecture. Discussions with David Garlan, Tom Mitchell and Jim Firby have clarified many of the issues involved in architectures for robot control. Thanks also to the comments of several anonymous reviewers.

References

[1] P. Agre and D. Chapman, "Pengi: An implementation of a theory of activity," in *Proc. National Conference on Artificial Intelligence*, Seattle, WA, 1987, pp. 268–272.

[2] J. S. Albus, "Outline for a theory of intelligence," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 21, no. 3, pp. 473–509, 1991.

[3] J. S. Albus, H. G. McCain, and R. Lumia, "NASA/NBS standard reference model for telerobot control system architecture," Technical Report 1235, National Institute of Standards and Technology, 1989.

[4] R. C. Arkin, "Motor schema based navigation for a mobile robot: An approach to programming by behavior," in *Proc. International Conference on Robotics and Automation*, Raleigh, NC, March 1987.

[5] R. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. 2 no. 1, 1986.

[6] R. Brooks, "A robot that walks: Emergent behavior from a carefully evolved network," *Neural Computation*, vol. 1, no. 2, pp. 253–262, Summer 1989.

[7] R. Brooks, "Intelligence without reason," in *Proc. International Joint Conference on AI*, Sydney, Australia, August 1991.

[8] L. Chrisman and R. Simmons, "Sensible planning: Focusing perceptual attention," in *Proc. National Conference on Artificial Intelligence*, Los Angeles, CA, July 1991, pp. 756–761.

[9] E. Clarke, E. Emerson, and A. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, April 1986.

[10] J. Connell, "A behavior-based arm controller," *IEEE Journal of Robotics and Automation*, vol. 5, no. 6, pp. 784–791, 1989.

[11] K. Dowling *et al.*, "Mobile robot system for ground servicing operations on the space shuttle," in *SPIE, Cooperative Intelligent Robotics in Space III*, Boston, MA, Nov. 1992, pp. 1829–1832.

[12] R. J. Firby, "Adaptive execution in complex dynamic worlds," Technical Report YALEU/CSD/RR 672, Yale University, 1989.

[13] R. J. Firby, "Building symbolic primitives with continuous control routines," in *AI Planning Systems*, College Park, MD, June 1992.

[14] T. Fong, B. Hine, and M. Sims, "Intelligent mechanisms group research summary," technical report, NASA Ames Research Center, Jan. 1992.

[15] R. Goodwin and R. Simmons, "Rational handling of multiple goals for mobile robots," in *First International Conference on AI Planning Systems*, College Park, MD, June 1992.

[16] B. Hayes-Roth, R. Washington, R. Hewett, M. Hewett, and A. Seiver, "Intelligent monitoring and control," in *Proc. International Joint Conference on AI*, Detroit, MI, August 1989, pp. 243–249.

[17] L. P. Kaelbling, "An architecture for intelligent reactive systems," in *Reasoning About Actions and Plans*, M. Georgeoff and A. Lansky, eds. San Matteo, CA: Morgan Kaufmann, 1987.

[18] A. Meystel, "Intelligent control in robotics," *Journal of Robotic Systems*, 1988.

[19] D. Olawsky and M. Gini, "Deferred planning and sensor use," in *Proceedings of DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, San Diego, CA, November 1990, pp. 166–174.

[20] K. Passino and P. Antsaklis, "A system and control theoretic perspective on artificial intelligence planning systems," *Applied Artificial Intelligence*, vol. 3, pp. 1–32, 1989.

[21] D. Payton, "An architecture for reflexive autonomous vehicle control," in *Proc. IEEE International Conference on Robotics and Automation*, San Francisco, CA, April 1986, pp. 1838–1845.

[22] S. Shafer, A. Stentz and C. Thorpe, "An architecture for sensor fusion in a mobile robot," in *Proc. IEEE International Conference on Robotics and Automation*, San Francisco, CA, April 1986, pp. 2002–2011.

[23] G. Shaffer and A. Stentz, "A robotic system for underground coal mining," in *IEEE Robotics and Automation Conference*, Nice, France, May 1992.

[24] R. Simmons, "Concurrent planning and execution for autonomous robots," *IEEE Control Systems*, vol. 12, no. 1, pp. 46–50, 1992.

[25] R. Simmons, "Monitoring and error recovery for autonomous walking," in *Proc. IEEE International Workshop on Intelligent Robots and Systems*, July 1992, pp. 1407–1412.

[26] R. Simmons, C. Fedor and J. Basista, "Task Control Architecture Programmer's Guide," Carnegie Mellon University, Robotics Institute, November 1992.

[27] R. Simmons and E. Krotkov, "An integrated walking system for the Ambler planetary rover," in *Proc. IEEE International Conference on Robotics and Automation*, Sacramento, CA, April 1991, pp. 2086–2091.

[28] R. Simmons, E. Krotkov, W. Whittaker, B. Albrecht, J. Bares, C. Fedor, R. Hoffman, H. Pangels, and D. Wettergreen, "Progress towards robotic exploration of extreme terrain," *Journal of Applied Intelligence*, vol. 2, pp. 163–180, 1992.

[29] R. Simmons, L. J. Lin and C. Fedor, "Autonomous task control for mobile robots," in *Proc. IEEE Symposium on Intelligent Control*, Philadelphia, PA, September 1990.

Reid Gordon Simmons is a Research Scientist in the Department of Computer Science and Robotics Institute at Carnegie Mellon University. He earned his B.A. degree in 1979 in Computer Science from SUNY at Buffalo, and his M.S. and Ph.D. degrees from MIT in 1983 and 1988, respectively, in the field of Artificial Intelligence. His thesis work focused on the combination of associational and causal reasoning for planning and interpretation tasks. The research analyzed the relationships between different aspects of expertise and developed a domain-independent theory of debugging faulty plans. Since coming to Carnegie Mellon in 1988, Dr. Simmons' research has focused on developing self-reliant robots that can autonomously operate over extended periods of time in unknown, unstructured environments. This work involves issues of robot control architectures that combine deliberative and reactive control, selective perception, and robust error detection and recovery. The ideas are currently being applied to a six-legged planetary rover, an indoor mobile manipulator, and an autonomous excavator. Dr. Simmons has published and lectured extensively in the area of architectures for autonomous mobile robots.