6056

AIAA-94-1210-CP

Task-Level Control for Autonomous Robots

N94-30561

Reid Simmons
School of Computer Science / Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
reids@cs.cmu.edu

267658
p. 7

Abstract

Task-level control refers to the integration and coordination of planning, perception, and real-time control to achieve given high-level goals. Autonomous mobile robots need task-level control to effectively achieve complex tasks in uncertain, dynamic environments. This paper describes the Task Control Architecture (TCA), an implemented system that provides commonly needed constructs for task-level control. Facilities provided by TCA include distributed communication, task decomposition and sequencing, resource management, monitoring and exception handling. TCA supports a design methodology in which robot systems are developed incrementally, starting first with deliberative plans that work in nominal situations, and then layering them with reactive behaviors that monitor plan execution and handle exceptions. To further support this approach, design and analysis tools are under development to provide ways of graphically viewing the system and validating its behavior.

Introduction

Most autonomous robot systems have specific tasks to perform — such as navigating to given locations, searching for particular objects, exploring the environment, etc. To make a robot perform its tasks reliably, it is desirable to provide explicit control over the achievement of tasks — controlling the sequencing of subtasks, monitoring their progress, handling exceptions, and managing the robot's limited computational and physical resources.

We refer to this as *task-level control:* the integration of planning, perception, and real-time control for the purpose of achieving high-level goals. To facilitate the development of task-level control systems, we have developed the Task Control Architecture (TCA). To date, TCA has been used in the development of about a dozen autonomous robot systems, including a walking rover [Simmons *et al.*, 1992], several indoor mobile robots [Simmons *et al.*, 1990], an excavator [Singh and Simmons, 1992], and an inspection robot for the Space Shuttle [Dowling and others, 1992].

The motivation for developing a task-level control architecture is that there appears to be a common set of control constructs that most autonomous mobile robots need, and that development of individual robot systems can be simplified by use of an architecture that explicitly supports those constructs. In much the same way as an operating system provides common facilities and hides details of the underlying computer, so too does TCA provide needed task-level control constructs while hiding details such as the mechanisms used for communication and task synchronization.

The facilities provided by TCA were chosen based on analysis of existing mobile robot systems and projected needs of future systems. The analysis showed that the architecture must facilitate the development of distributed, modular, and concurrent systems. In addition, a task-level control architecture should allow the concurrency to be controlled in a selective (and explicit) manner, so that distributed processes do not interact in undesirable ways. This includes providing methods for sequencing and synchronization of subtasks, as well as managing access to system resources (e.g., cameras, actuators, computers). Finally, to cope with uncertainties in the environment and uncertainties in the achievement of subtasks, the architecture needs to support extensive, task-dependent monitoring and exception-handling strategies.

In addition to providing all the above capabilities, the Task Control Architecture supports a particular methodology for designing and developing autonomous robot systems. The approach, which we term *structured control*, involves first developing basic deliberative components that handle nominal situations, and then increasing reliability by incrementally layering on reactive behaviors to handle exceptions. With TCA, this can be done without requiring significant modification to the existing robot software system. In particular, monitors and exception handlers can be added after the

[1]Copyright ©1993 by Reid Simmons. Published by the American Institute of Aeronautics and Astronautics, Inc. with permission

basic system has been developed.

This layering of reactive behaviors on to a deliberative base provides an engineering basis for developing autonomous mobile robot systems. First, incomplete understanding of the tasks, environment or hardware is accommodated by separating the design into nominal, and presumably better understood, behaviors and the more numerous, but infrequently occurring, exceptional situations (which may become known and understood only during testing of the robot system). Second, the separation of nominal and exceptional behaviors increases overall system understandability by isolating different concerns: the robot's behavior during normal operation is readily apparent, and strategies for handling exceptions can be developed separately and then added to the existing system with a minimum of effort. Finally, complex interactions are minimized by constraining the applicability of reactive behaviors to specific situations, so that only manageable, predictable subsets of the behaviors will be active at any one time.

The rest of this paper describes the Task Control Architecture in more detail, focusing on a few applications of the architecture to the development of autonomous mobile robot systems. The paper concludes with a brief description of where the development of TCA is heading — in particular, describing design and analysis tools that we are beginning to develop.

The Task Control Architecture

The Task Control Architecture has been designed to facilitate the process of developing and controlling autonomous robot systems that must perceive, plan and act in uncertain, dynamic environments [Simmons, 1992a, Simmons, 1992b]. TCA provides a language for expressing task-level control decisions, and provides software utilities for ensuring that those control choices are correctly realized by the robot. The five major types of control constructs supported by TCA are:

- distributed communication
- task decomposition and sequencing
- resource management
- execution monitoring
- exception handling

A system built using TCA consists of robot-specific processes (called *modules*) that communicate by sending messages via a general-purpose *central control module* (see Fig. 1). Modules can be written in either C or LISP, and can operate on a number of different computer platforms (including Sun, SGI, Vax, MacIntosh, and 680xx and i486 processors) and on different operating systems (including Unix, VxWorks and Mach).

The robot-specific modules register with the central control module which messages they can handle, along with the data formats associated with the messages. The data formats can be complex, including embedded structures, arrays, and pointers. TCA is responsible for encoding and decoding the data into byte streams and routing messages (via sockets) to the appropriate

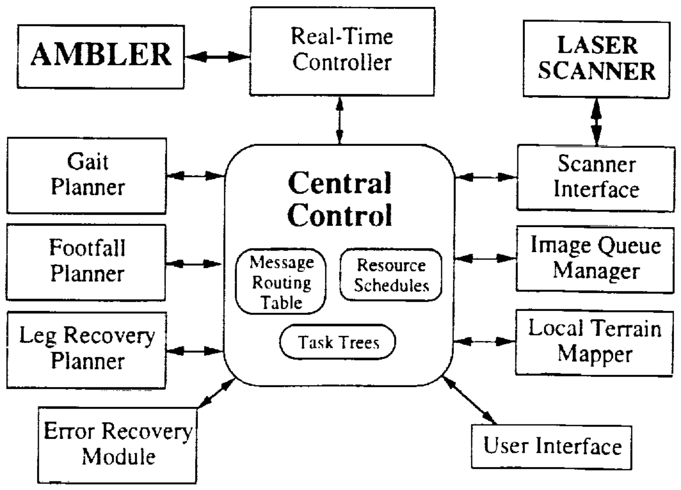

Figure 1: Task Modules for Ambler Walking System

modules to be handled. Messages are *anonymous*, that is, the sending and receiving modules do not know each other's identities. This facilitates modular development — one module can easily be substituted for another with the same functionality (even while the rest of the system continues to operate). Thus, for example, a graphical simulator that has the same message interface as the real-time controller can be substituted at will, which greatly facilitates the development and debugging process (as well as protecting valuable robotic hardware!).

TCA provides different types of messages, each with somewhat different semantics. For example, *inform* messages provide one-way communication between processes; *query* messages provide two-way communication (providing a client-server relationship), and *broadcast* messages enable one module to distribute data to any number of receiving modules simultaneously. Other message types, including goals, commands, monitors and exceptions, will be discussed below.

Task Decomposition

Besides providing for data communication, TCA provides a host of facilities for coordinating robot systems at the task level. Modules use the TCA control constructs to constrain the robot's behavior. For example, a module can specify the order in which subtasks should be carried out, or indicate when and how to monitor for exceptional conditions.

Central to TCA is a hierarchical representation of subtasks called *task trees*. In essence, a task tree is TCA's notion of a plan, representing both goal/subgoal decomposition, as well as *temporal constraints* between node, which indicate (partial) orderings on their execution. TCA constructs and maintains task trees dynamically: nodes in the task tree are associated with

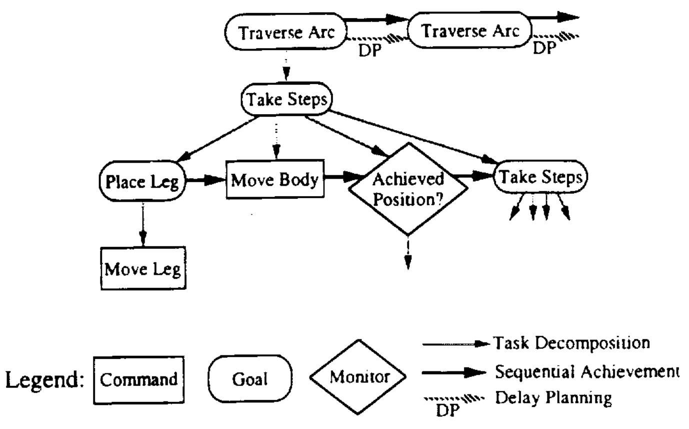

Figure 2: Task Tree for Ambler Walking

messages; when a message handler itself issues a message, a child is added under the node associated with the message being handled. TCA utilizes the subgoal and temporal constraint information to schedule and coordinate the sending of messages.

Figure 2, for instance, illustrates a simplified version of the task tree for autonomous walking of the Ambler rover [Simmons *et al.*, 1992]. In the figure, narrow vertical arrows denote task decomposition and heavy horizontal arrows denote temporal constraints on task planning and execution. The task tree indicates that the Ambler sequentially traverses a series of arcs, where planning how to traverse one arc is delayed until the previous arc has been completely traversed. Traversing an arc consists of taking a sequence of steps, with each step consisting of a pair of leg and body moves. Unless the end of the arc has been reached, the planning module handling the "Take Steps" message recursively issues another "Take Steps" message. Note that the absence of a ***delay planning*** (DP) temporal constraint between the "Achieved Position?" monitor node and subsequent "Take Steps" goal node indicates to TCA that planning one step can occur concurrently with the execution of the previous step. This use of concurrency enables the Ambler to achieve nearly continuous motion [Simmons, 1992a].

Resource Management

Many robot systems have limited resources that must be managed efficiently. This is particularly important when the robot system consists of multiple, interacting processes in order to prevent resource contention and conflict. For example, if the robot has a camera on a pan/tilt head, the processes that need visual information must have ways to point the camera and to ensure that no other process will re-aim it until the required images have been acquired. Similarly, a robot system might want to ensure that a planning module remained available to deal with an upcoming, high priority request.

TCA provides support for this type of resource management. Procedures that handle messages can be grouped into logical units, called ***resources***. These units can, in turn, be grouped into modules (see, for instance, Fig. 1). TCA maintains the constraint that only one message will be handled by a resource at a time. However, since modules may consist of multiple resources, a module can be processing multiple messages at once (for instance, if it is running in a multi-tasking environment such as VxWorks). This division into resources and modules is totally up to the discretion of the robot system designer, and can be organized so as to promote modularity, efficient use of resources, or the need to access a common piece of hardware.

TCA also enables a module to ***lock*** the resource of another module. This prevents any other module from accessing the resource until it is unlocked. This provides a mechanism for synchronizing subtasks: the resource can be locked while a time-critical operation is taking place, and then unlocked to enable normal message flow. In the Ambler system, for example, the perception module locks the real-time controller resource before acquiring laser range images, in order to prevent blurring.

Monitoring and Exception Handling

One of the most important task-level control functions for an autonomous mobile robot is to monitor its progress and safety, and to handle exceptions arising from violated expectations. The structured-control approach to designing robot systems advocates that such reactive behaviors be added incrementally, on top of the task tree that represents the basic, deliberative plan for achieving the task.

The rationale here is that, for complex tasks and environments, it is too difficult to design a system from the start that acts correctly in all situations. This is primarily because either the environment is not that well understood (especially if it is dynamic or remote, such as the surface of another planet) or the interactions between the environment and the robot are not well understood (such as for an excavation robot). Often the best that can be done in such cases is to design for the known situations first, and then incrementally debug and extend the system as experience dictates.

TCA provides several mechanisms that directly support this approach. For one, exception handling strategies can be added incrementally without modifying existing components: a module can add information to an existing task tree to indicate which procedures TCA should invoke in response to exceptions raised by other modules. When an exception is raised, TCA searches up the task tree to find a handler designated for that exception. If the exception handler finds it cannot actually deal with the particular situation, it reissues the exception and the search continues up the tree. Typically, the strategies for dealing with exceptions involve

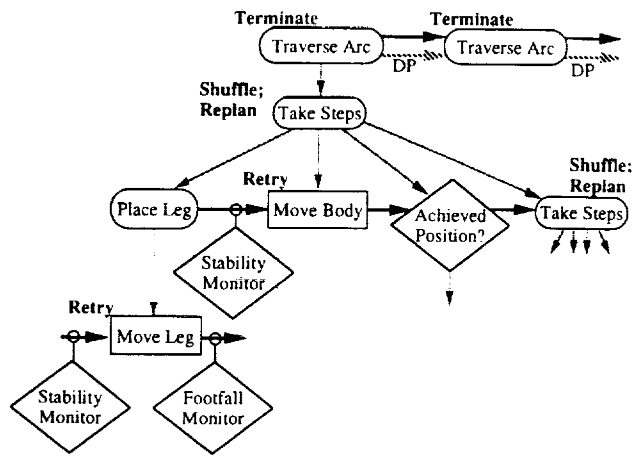

Figure 3: Task Tree with Monitors and Exception Handlers

modifying the currently executing plan, either by killing off parts of the task tree or adding new nodes and/or temporal constraints to the tree.

For example (Fig. 3), the Ambler real-time controller monitors force sensors in the feet and raises an exception when a threshold is exceeded (indicating unexpected terrain contact). A separate error recovery module handles this by modifying the current leg trajectory to surmount the obstacle, and then instructs TCA to re-execute the trajectory [Simmons, 1992b]. If modifying the leg trajectory fails to clear the obstacle, the complete move may be replanned, the Ambler's feet may be shuffled into a standard configuration, etc. Ultimately, if no fix is found, the walking task is terminated and the user is notified.

Just as it makes sense to take advantage of hierarchy in decomposing tasks into subtasks, it makes sense to treat exceptions in a hierarchical fashion. The idea is that lower-level exception handlers are more specific to a given failure, and can have more local, direct effects on the problem; the handlers located higher up the tree handle a wider range of exceptions, but since their effects are broader and have more impact on the overall plan, they should be tried only when the more specific strategies fail.

Execution monitors can also be added incrementally using the TCA *wiretap* control construct. The wiretap mechanism enables a monitor to be associated with a class of messages, so that the monitor is automatically triggered whenever a message of that class is handled. For example, before every leg or body move of the Ambler, a stability monitor is invoked to verify that the move will not cause the robot to tip over; after every leg move, a footfall monitor analyzes the force sensor data to detect possibly unstable footholds (see Fig. 3).

These monitors were added after the basic walking component of the Ambler was designed and debugged, in order to enable the system to handle increasingly difficult terrain and longer distances. For example, in one experiment, the Ambler walked over 500 meters outdoors in hilly terrain (with slopes up to 30%). During the experiment, in which the Ambler took over 1000 footsteps, many exceptional situations were encountered: unexpected terrain collisions, hardware faults (amplifiers, motion faults, sensor failures) and software faults (mainly when the planners could not find suitable footfalls). All these situations were dealt with by the robot itself: the conditions were detected in a timely manner and, except for certain hardware faults where humans had to manually reset the hardware, the robot autonomously recovered from the situations and continued walking.

Monitors can also be added to check for ongoing opportunities or contingencies. For example, one of our indoor mobile robots has the task of keeping the lab floor free of cups [Simmons *et al.*, 1990]. The robot system employs one monitor to check whether a new cup has been spotted by the vision system. For every cup found, a goal is added to retrieve the cup and another monitor is added which checks to ensure that the cup is still visible. If the cup disappears from view, then it is assumed that someone else picked it up, and the monitor cancels the associated "cup retrieval" task. Thus, the system is able to handle multiple goals that are both activated and deactivated asynchronously.

Comparisons

TCA and the structured-control approach differ from the *behavior-based* approach, in which systems consist of collections of local behaviors that act according to direct sensing of the environment [Brooks, 1986, Connell, 1989]. The global behavior of such systems typically emerge from interactions between the local behaviors [Agre and Chapman, 1987, Brooks, 1991]. A problem with the behavior-based approach is it assumes that robust primitive behaviors can be developed that act correctly in all, or most, situations. This can be very difficult in practice, given incomplete knowledge about the environment and the robot's interaction with it. In contrast, the structured-control approach advocates developing complete components for limited environments, and incrementally updating the design to handle more challenging and diverse requirements.

The approach also differs from other hierarchical architectures, such as NASREM [Albus *et al.*, 1989], in which the flow of control is primarily top-down. While top-down task decomposition is important in TCA, the architecture also provides for significant bottom-up control in its use of monitors and hierarchically scoped exception handlers. This enables autonomous robot systems to be very reactive to changes in the environment.

The approach used in TCA is probably closest in flavor to the RAPs system [Firby, 1989] and related architectures [Gat, 1992, Georgeff and Lansky, 1987],

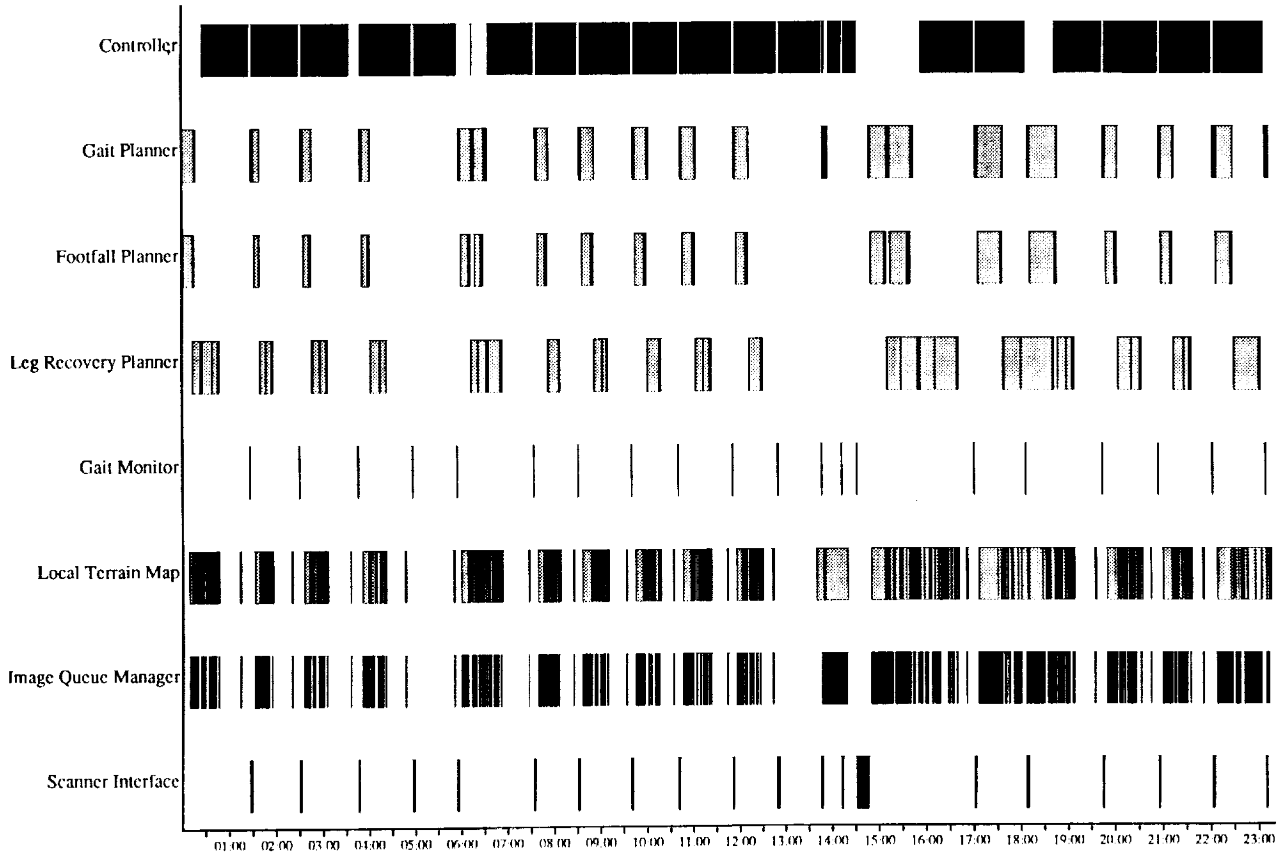

Figure 4: Gantt Chart of Module Activity

which feature temporal sequencing of subtasks in conjunction with monitoring and error recovery. The main differences are that TCA is based on true concurrency, rather than interleaving of subtasks (which allows it to exhibit better real-time performance), and that planning, monitoring and exception handling are all cleanly separated (which facilitates evolutionary robot system development).

Design and Analysis Tools

While TCA and the structured-control approach have proven useful for complex, autonomous robot systems, in practice developing such systems is often a time-consuming, trial-and-error process. To reduce this effort, we are currently developing tools to aid in the analysis and design of TCA-based robot systems.

The first two tools that we developed analyze the log files that TCA produces of all message traffic. The log files contain important information regarding the types and order of messages sent within the system. One tool in current use processes log files and produces graphical representations of TCA task trees (similar to that shown in Fig. 2). A developer can recreate the task tree message by message, either *post hoc* or as the system runs, to see what the task tree looks like as it evolves, and what temporal interactions might be causing problems. This tool has proven particularly valuable because it is typically difficult to predict in advance the behavior of complex distributed systems due to subtle timing interactions between processes.

Another tool analyzes log files to produce Gantt charts showing module activity (see Fig. 4 — the dark bars indicate when a module is processing messages; the light bars indicate when it is waiting for the reply to a query message). For each module, the chart shows which messages it is processing at what times, and when messages are queued due to resource contention. This tool has been used to find bottlenecks in system performance. For example, it was used in the development of the Ambler system to determine how to maximize performance through the use of concurrency. The Ambler system was originally developed with a sequential sense-plan-act cycle. The use of this tool indicated that continuous motion could be obtained by executing one step while planning the next one, since the time needed for executing steps exceeded the planning time for steps [Simmons, 1992a]. More recently, a similar analysis indicated that perception was the bottleneck in system performance: based on this, TCA control constructs were added to make some of the perceptual processing concurrent, as well [Hoffman and Krotkov, 1991].

We are beginning development of additional tools to aid in the design of mobile robot systems. One tool, similar in spirit to a CASE tool, would enable designers to graphically specify task decomposition strategies, including conditionals, loops, temporal constraints, monitors and exception handlers. The tool would then generate the TCA calls needed to implement those specifications. We anticipate that this tool will be very useful in rapidly prototyping system designs and in documenting the design process.

Eventually, we would like for the tool to actually help validate the system design, detecting problems such as malformed data interfaces between modules, potential deadlock situations, resource contention, etc. To do this, we need to apply automated reasoning techniques to TCA-based system designs (for instance, using model-checking techniques [Clarke *et al.*, 1986]). To this end, we have begun formalizing the Task Control Architecture control constructs using a combination of temporal logic and the Z notion [Spivey, 1992].

For example, the following schemas give the basic formalization of the notion of task trees: a task tree is a set of nodes, each of which has a parent. The "received" set consists of the messages that TCA has received and the "finished" set contains those that have already been handled by some module. A task tree node, in turn, has an associated handler, type, and state (received, running, finished) and a set of temporal constraints. The task tree schema places some conditions on the temporal constraints of various nodes of the task tree.

$$
\begin{array}{l}
\textit{TaskTree} \\
\hline
\textit{nodes} : \mathbb{P}\,\textit{Node} \\
\textit{parent} : \textit{Node} \nrightarrow \textit{Node} \\
\textit{received} : \mathrm{seq}\,\textit{Node} \\
\textit{finished} : \mathrm{seq}\,\textit{Node} \\
\hline
\forall\, \textit{node}, \textit{node2} : \textit{Node} \bullet \\
\quad (\textit{node.type} \in \{\textit{Query}, \textit{Inform}\} \Rightarrow \\
\qquad \textit{parent}(\textit{node}) = \textit{root} \land \\
\qquad \textit{node.achievConst} = \varnothing \land \\
\qquad \textit{node.onHoldUntil} = \varnothing) \land \\
\quad (\textit{parent}(\textit{node}) = \textit{node2} \Rightarrow \\
\qquad \textit{node.achievConst} \subseteq \textit{node2.achievConst}) \land \\
\quad \textit{node.handler} = \textit{node2.handler} \Leftrightarrow \\
\qquad \textit{node} = \textit{node2} \\
\textit{root} \notin \textit{nodes} \\
\textit{nodes} = \mathrm{ran}\,\textit{parent} \land \mathrm{dom}\,\textit{parent} = \textit{nodes} \cup \{\textit{root}\} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{Node} \\
\hline
\textit{handler} : \textit{HANDLER\_ID} \\
\textit{type} : \textit{NODE\_TYPE} \\
\textit{state} : \textit{EXECUTION\_STATE} \\
\textit{achievConst} : \mathbb{P}\,\textit{TEMPORAL\_CONSTRAINT} \\
\textit{onHoldUntil} : \mathbb{P}\,\textit{TEMPORAL\_CONSTRAINT} \\
\hline
\textit{type} = \textit{Command} \Rightarrow \\
\quad \textit{achievConst} = \textit{onHoldUntil} \\
\textit{type} \in \{\textit{Query}, \textit{Inform}\} \Rightarrow \\
\quad \textit{achievConst} = \varnothing \land \\
\quad \textit{onHoldUntil} = \varnothing \\
\hline
\end{array}
$$

When the formalizations are completed, we expect to use them to prove properties about the performance of specific robot systems. For example, using the current temporal formalization, we can show that the temporal constraints described in [Simmons, 1992a] are sufficient to ensure that the Ambler walking system will plan at most one step in advance. We would also like to use the Z formalization to prove the correctness of the implementation of TCA, to give users confidence that the architecture correctly meets the intended semantics.

Conclusions

Autonomous robot systems need task-level control in order to effectively integrate planning, perception and actuation to perform complex tasks in uncertain, dynamic environments. The Task Control Architecture (TCA) has been developed to facilitate the creation of task-level control systems. TCA provides control constructs that are commonly needed by autonomous robot systems, including distributed communication, task decomposition and sequencing, resource management, monitoring and exception handling,

TCA supports the *structured-control* methodology of system development in which plans are first designed to work in nominal situations, and then reactive behaviors (execution monitors and exception handlers) are layered on to the base of deliberative plans. We argue that such a design philosophy is useful in situations where the environment the robot will be operating in, and/or the robot/environment interactions, are not totally understood.

It is our contention that reliable performance in a wide range of situations can best be obtained by incrementally adding on reactive behaviors that deal with specific, previously unanticipated, situations. It is also beneficial to structure such behaviors hierarchically, relying first on lower-level reactions that have specific, but local, effects, and using higher-level reactions with more global effects only when the more specific ones fail to solve the problem.

TCA and the structured-control design methodology have been used in developing about a dozen autonomous mobile robots, including a planetary rover, an indoor mobile manipulator, an excavator, and a robot for inspecting the Space Shuttle. In each case, the communication and control constructs provided by TCA made it easier to develop and debug the concurrent, distributed systems.

We are continuing our efforts by providing design and analysis tools to support the development of TCA-based systems. In particular, we are formalizing the TCA control constructs in order to provide tools for automatically reasoning about and validating system designs.

Acknowledgments

Many thanks to Christopher Fedor, Long-Ji Lin and Richard Goodwin who implemented much of TCA, and members of the Planetary Rover project who have developed the Ambler and its autonomous walking system. Domingo Gallardo did the Z formalization of TCA. Thanks also to David Garlan for many useful ideas and discussions. This research is supported in part by NASA under Contract NAGW-1175 and by OSD under Contract F08635-92-C-0019.

References

[Agre and Chapman, 1987] Phil Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proc. National Conference on Artificial Intelligence*, pages 268–272, Seattle, WA, 1987.

[Albus *et al.*, 1989] James S. Albus, Harry G. McCain, and Ronald Lumia. NASA/NBS standard reference model for telerobot control system architecture (NASREM). Technical Report 1235, National Institute of Standards and Technology, 1989.

[Brooks, 1986] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1), 1986.

[Brooks, 1991] Rodney Brooks. Intelligence without reason. In *Proc. International Joint Conference on AI*, Sydney, Australia, August 1991.

[Clarke *et al.*, 1986] Edward Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[Connell, 1989] Jonathan Connell. A behavior-based arm controller. *IEEE Journal of Robotics and Automation*, 5(6):784–791, 1989.

[Dowling and others, 1992] Kevin Dowling et al. Mobile robot system for ground servicing operations on the space shuttle. In *SPIE, Cooperative Intelligent Robotics in Space III*, pages 1829–1832, Boston, MA, November 1992.

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

[Gat, 1992] Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proc. National Conference on Artificial Intelligence*, pages 809–815, San Jose, CA, July 1992.

[Georgeff and Lansky, 1987] Mike Georgeff and Amy Lansky. Reactive reasoning and planning. In *Proc. National Conference on Artificial Intelligence*, pages 972–978, Seattle, WA, July 1987.

[Hoffman and Krotkov, 1991] Regis Hoffman and Eric Krotkov. Perception of rugged terrain for a walking robot: True confessions and new directions. In *Proc. of the International Workshop on Intelligent Robots and Systems*, pages 1505–1510, Osaka, Japan, November 1991.

[Simmons *et al.*, 1990] Reid Simmons, Long Ji Lin, and Chris Fedor. Autonomous task control for mobile robots. In *Proc. IEEE Symposium on Intelligent Control*, Philadelphia, PA, September 1990.

[Simmons *et al.*, 1992] Reid Simmons, Eric Krotkov, William Whittaker, et al. Progress towards robotic exploration of extreme terrain. *Journal of Applied Intelligence*, 2:163–180, 1992.

[Simmons, 1992a] Reid Simmons. Concurrent planning and execution for autonomous robots. *IEEE Control Systems*, 12(1):46–50, February 1992.

[Simmons, 1992b] Reid Simmons. Monitoring and error recovery for autonomous walking. In *Proc. IEEE International Workshop on Intelligent Robots and Systems*, pages 1407–1412, July 1992.

[Singh and Simmons, 1992] Sanjiv Singh and Reid Simmons. Task planning for robotic excavation. In *Proc. IEEE Conference on Intelligent Robots and Systems*, Raleigh, NC, July 1992.

[Spivey, 1992] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, second edition, 1992.