Towards Reliable Autonomous Agents

Reid Simmons
School of Computer Science/Robotics Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15241
(reids@cs.cmu.edu)

Abstract

We are interested in producing reliable autonomous robots that can operate for extended periods of time in uncertain, dynamic environments. We have been developing methodologies and software tools to facilitate this, including the Task Control Architecture and probabilistic methods for representing and reasoning about uncertainty. The aim is to incrementally produce reliable behavior by adding (perhaps automatically) execution monitors and exception handlers that can detect when expectations are being violated and react appropriately. We have been developing and testing these ideas on a number of mobile robot platforms, including a six-legged planetary rover, a wheeled Lunar rover, and an office-navigation robot.

Introduction

Reliability is a key aspect of autonomous agents, especially those that operate in uncertain and dynamic environments. They need to know their capabilities and limitations, when they are getting into trouble, and how to recover from exceptional situations. We have been developing methodologies and software tools to facilitate the development of reliable mobile robotic systems. This includes an architecture (the Task Control Architecture) that combines deliberative and reactive control and that enables the incremental addition of task-specific execution monitors and exception handlers. It is our contention that such a capability is essential, given that one cannot hope to determine, *a priori*, all the things that could go wrong with an autonomous agent. Instead, we desire to equip robots with the ability to detect general lack of progress toward their goals, and with enough domain knowledge to learn how to monitor for specific anomalous situations and recover from them.

Our methodology has several components. First, we have developed the Task Control Architecture (TCA), a general-purpose architecture for building distributed, concurrent robot systems. TCA combines both deliberative and reactive control in a paradigm we call *structured control* [9]. The idea is that one starts with deliberative (planned) behavior that is expected to work correctly in nominal situations, and then adds reactive behaviors (in particular, monitors and exception handlers) to deal with exceptional situations. We believe (and our experience to date bears this out [6, 8]) that the separation of nominal and exceptional behaviors increases system reliability. In particular, understandability is increased by isolating different concerns: the robot's behavior during normal operation is readily apparent, and strategies for handling exceptions can be developed in isolation. Furthermore, 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.

Another aspect of our methodology is to make the robot explicitly aware of why it is pursuing particular goals. This knowledge is supplied in the form of *expectations* about the likely/desired results of actions [7]. For example, depending on the type of terrain it is crossing, a Lunar rover might expect its inclinometers to indicate level, steady ground, or pitched, rough terrain. If the readings fall outside of the expectations, then it is an exceptional situation and must be attended to [4]. The idea is that these expectations provide a context for interpreting sensor readings. This stands in contrast to more purely reactive architectures, in which the behaviors are always active, and so the way sensor readings are interpreted is largely context-independent. We believe that the use of expectations provides an added measure of reliability, since the robot can now explicitly reason about its progress (or lack thereof) towards achieving its goals.

A third aspect of our methodology is that robotic systems need to be developed incrementally. In particular, the reactive part of the system, to detect and handle exceptional situations, must be layered on to the deliberative part in an evolutionary fashion. This stems from the belief that the ways robots interact with their environments are too complex and uncertain to analyze completely beforehand. The best that can be done is to start with a plan of action that usually works, along with some reactive strategies that can

be derived from a known understanding of the environment, and then to add new reactive strategies as experience dictates (whether that means adding by hand, or through automated learning). We have had positive experience with this methodology. For example, for a walking robot we added successive monitors and exception handlers that enabled the robot to handle increasingly rugged terrain without tipping over [6]. For an indoor office-navigation robot, we added monitors and exception handlers that increased the success rate of the robot in traversing corridors [8]. These strategies helped the robot deal with sensor noise, dead-reckoning inaccuracies, and even topological differences between its map and the actual environment.

The Task Control Architecture

The Task Control Architecture (TCA) was designed to facilitate the development of task-level control systems for autonomous and semi-autonomous robots operating in uncertain, dynamic environments [5, 6, 9]. The term *task-level control* refers to the problem of coordinating perception, planning, and actuation to achieve a given set of goals. TCA provides a language for expressing task-level control decisions, and software utilities for ensuring that the decisions are correctly realized. In essence, TCA is a high-level robot operating system that provides an integrated set of commonly needed control constructs, including distributed communications, task decomposition, task sequencing, resource management, execution monitoring, and exception handling.

A robot system built using TCA consists of a number of distributed, robot-specific modules (processes) that communicate by sending messages. The modules specify control information by indicating how to decompose tasks into subtasks, when to monitor for specific situations, and how to handle exceptions. TCA provides a library of communication tools and a robot-independent *central control module* that is responsible for routing messages and maintaining the task control information specified by the modules. In particular, TCA maintains a hierarchical *task tree* (Figure 1) that represents the robot's intended plan of action. The task tree is used, in turn, to schedule and coordinate the actions of the other modules. The idea is that all modules operate concurrently, by default, unless control information specifies otherwise. TCA has been used to control about a dozen robotic systems at CMU and elsewhere, in particular a six-legged planetary rover [10], a Lunar rover [3], and an autonomous indoor mobile robot [8].

TCA provides several mechanisms for integrating deliberative and reactive control. Support for deliberative control is provided by constructs for decomposing tasks into subtasks and for scheduling and sequencing subtasks, which TCA uses to create its task trees. A novel feature of TCA is that the planning (task decomposition) of tasks can also be scheduled via temporal constraints on nodes in the task tree. This provides a flexible way to encode interleaving of planning and execution.

Support for reactive control is provided by constructs for monitoring and handling exceptions. TCA monitors consist of a trigger condition, and an action that is performed whenever the trigger condition is perceived (for example, the robot levels itself whenever its inclinometers exceed a given threshold). Monitors are context-dependent: they are constrained to operate concurrently with particular subtasks, and they cease to be active when the associated subtasks complete. This enables the robot systems to utilize resources most efficiently, for instance, by not monitoring for situations that do not apply in the current context (e.g., not looking for hallway features when trying to pick up a piece of trash).

Exception handlers are also context-dependent, and are maintained in a hierarchy. This is done by associating separate exception handlers with nodes in the task tree. When an exception is raised, TCA searches up the task hierarchy to find the first exception handler that matches the given exception. If that handler cannot, in fact, handle the exception, the search continues up the tree. In this way, local exception handlers can be tried first (which, presumably, make minimal changes to the existing plan), but if those strategies fail other, more global, exception strategies can be tried. Exception handlers typically act by modifying the existing plan (the task tree) by adding, removing, or rearranging subtrees [6, 9]. For example, if a leg move fails, a low-level exception handler might retry the motion; a higher-level exception handler can replan the trajectory; and an even higher-level handler might give up and try achieving a different goal altogether (see Figure 1).

Both monitors and exception handlers can be added incrementally, even while the robot is running, without the need to change the existing system. This enables reliability to be added incrementally, as the need arises. It is our contention that it is difficult, if not impossible, to pre-specify all the exceptional situations that will be relevant to an autonomous robot. The best that can be done is to define general purpose monitors (e.g., progress monitors, proximity detectors) that will prevent the robot from injuring itself or others. Then, on a case-by-case basis, additional monitors and exception handlers can be added to handle specific cases that arise in practice [8]. While, currently, we add such reactive components by hand, we are investigating techniques for learning such strategies, based on the robot's expectations of its behavior and its experiences in the world.

Testbeds

While many TCA-based robot systems have been developed, to date we have used three main testbeds for developing our ideas. The Ambler is a six-legged robot developed for planetary exploration of very rugged ar-

Figure 1: Task Tree for a Walking Rover

eas [10]. While it has a great deal of mobility, it is also somewhat unstable. Thus, in addition to making it walk, we have to be careful not to let it tip over. The main problem is that our terrain information (garnered from a laser range scanner) is noisy, and so sometimes the robot steps on rocks, from which it might slip off. Thus, the architecture we developed included a planning algorithm that usually found acceptable footfalls for the robot, and a collection of monitors that looked for exceptional situations (mainly tilt readings and force readings in the feet that indicated imminent slippage). In addition, we added specialized heuristics for getting out of situations that the main planner could not handle. For example, it was usually possible to move the feet into a standard configuration, after which the nominal planning algorithm could find a good footfall.

Another major testbed is the Ratler, a wheeled Lunar rover that was originally developed at Sandia National Laboratories (Figure 2). The Ratler is a prototype rover for a semi-autonomous lunar mission that will feature long-term operation and will be driven by novice users [3]. To achieve these goals, the rover must be extremely reliable, able to detect hazardous situations, and prevent users from driving the rover into danger. In addition, the rover needs to monitor its own health and safety, performing many diagnostic and fault recovery procedures on its own. TCA is being used as the underlying architecture for the rover, with specific attention being paid to issues of monitoring internal and external environmental conditions.

The third main testbed is Xavier, an indoor mobile robot that autonomously navigates the corridors of our building (Figure 3). Xavier is a synchro-drive robot with bump, sonar, laser, and camera sensors. It mainly uses landmark-based navigation, where landmarks are doorways, corridor intersections, and walls in front of it (although, we have recently implemented a more reliable probabilistic navigation scheme). Since the landmark detector is not completely reliable, the robot occasionally misses a landmark and gets lost (sometimes the landmark itself disappears, for instance, when people are standing in front of an open doorway). Xavier has a number of reactive strategies for dealing with such problems, including searching more carefully the area in which the landmark should have appeared, turning around if necessary to get another look [8]. Current plans include having Xavier navigate all day in the building, performing delivery tasks and recycling tasks (an arm will be added to Xavier this Winter).

Current Work

While we are relatively happy with the overall architecture and development methodology, there are still a number of technology gaps that we are working on. Mainly, these relate to two problems: 1) ensuring that the concurrent reactive behaviors do not interact badly, and 2) determining what new reactive strategies to add.

The first problem arises because, even though we impose a structure on when monitors and other behaviors are active, there are still behaviors that operate concurrently. The problem can be exacerbated by incremental system development, since there is more chance for the behaviors not to share common assumptions (and, hence, interact badly) if they are developed at separate

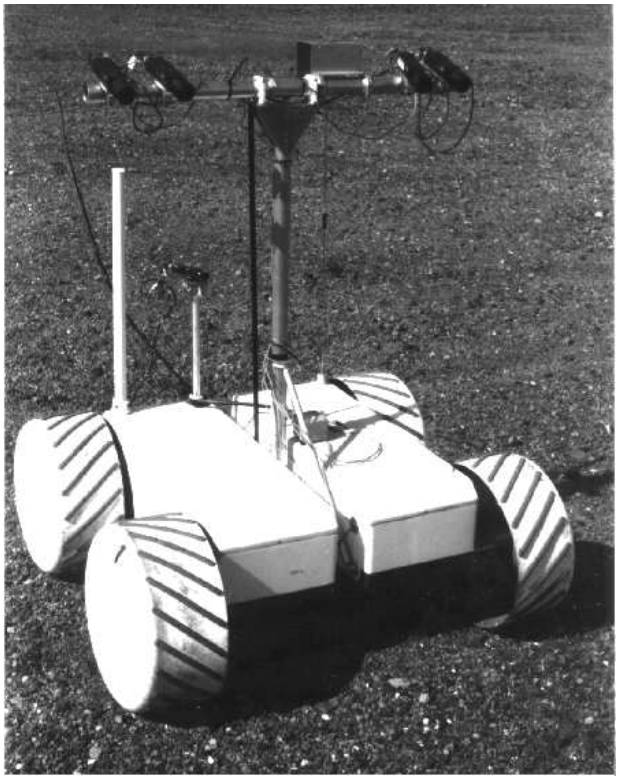

Figure 2: Ratler Lunar Rover

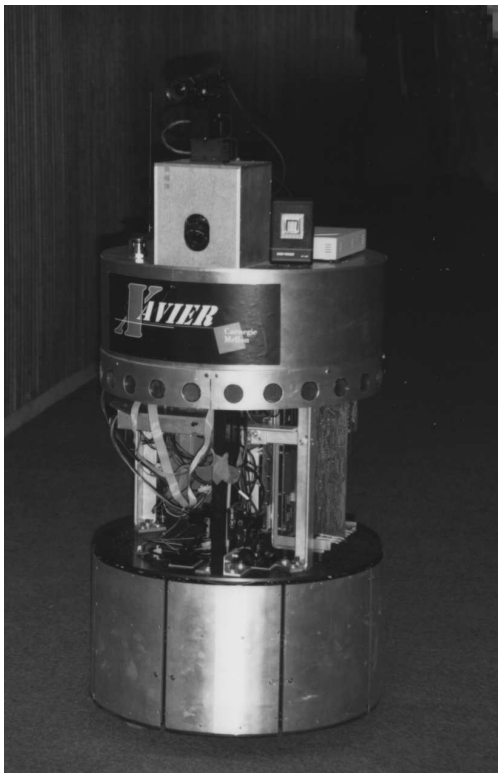

Figure 3: Xavier Mobile Robot

times. Our approach here is to attempt to use formal languages to model the systems and their interactions, and to use analysis tools to determine attributes of the system, such as the potential for deadlock, resource contention, etc. We have been looking at various modeling languages, such as Petri Nets, Z, state charts, and temporal logic to model TCA-based systems. While this work is still in its infancy [1], we believe that it is a fruitful direction to explore.

Along the same vein, we are developing software tools to help automate the design, and analyze the performance, of TCA-based systems. The design tools will take abstract, high-level descriptions of the communications interface and task decomposition strategies of a robot system and automatically generate code, or code stubs. These tools will include a semantic component to catch obvious errors in interface specifications and subtask interactions. The analysis tools will analyze logs of the TCA message traffic and graphically display pertinent information, such as resource utilization, scheduling constraints, and task interdependencies. Currently, we have prototypes of an interface definition tool and two analysis tools to chart resource utilization and task-tree dependencies.

The second problem is more fundamental: where do the monitors and exception handlers come from? We want to move from hand-coding them to learning such reactive strategies based on experience. The idea is that one would have general progress monitors that would be good at detecting when things were really going badly (such as noticing that the robot has bumped something, or that it is not moving forward, as expected). The problem with such monitors is that they are too coarse – they eventually catch all problems, but not necessarily in time to allow the robot to extricate itself easily. Thus, we would like more task-specific, finer-grained monitors that are tuned to the expectations inherent in the particular task, or subtask, currently being performed. When one of the general-purpose monitors trigger, it indicates that no more specialized monitor exists for that situation, and thus it presents an opportunity for learning. The vision is that the robot would use the actual experience it is in, plus the plan it was pursuing, to determine what it was expecting to sense in that situation, and from that to determine what it *could have* sensed to predict that it would be getting into that situation (and, presumably, take action to avoid getting into that situation).

Before we can move in that direction, however, we feel that we must move from representations and plans that are mainly discrete and deterministic, to more probabilistic representations and more conditional plans. This will enable us to reason more robustly about what went wrong and why it did so (for instance, it is often difficult to determine whether the root cause of a problem is sensor, planning, or execution error). In addition,

we have found that it is often hard to make rational planning decisions (e.g., deciding where to place a foot, or when the robot is lost) when the sensor data and world models are so fraught with uncertainty. While we do not wish to give up on the power of symbolic representation and planning, we also do not want to use it when inappropriate.

We present two examples of this. We originally developed a symbolic, landmark-based navigation scheme for Xavier [8]. It navigates by planning a path (using A*) through a node-and-arc topological map of the environment. Based on the map, it determines which landmarks it expects to find at the intersections where it wants to turn. While, as described above, there are a number of monitors and exception handlers for situations where it makes a mistake, it still has only a discrete idea of where it is at any one time, and so must be fairly conservative before it decides that it has truly made a mistake. In contrast, we have recently developed a navigation system for Xavier that uses partially observable Markov models to explicitly represent the robot's uncertainty in its position. When the robot gets a new sensor reading, it uses an uncertainty model of its sensors to determine how likely it is to have observed that reading at various places in the map, and conditions that with the probability that it currently believes that it is at that place. A planner associates different actions with each Markov node, and the robot continually executes the action with the largest total probability mass.

In our simulation tests, this method has worked quite well, even with relatively noisy sensor models (on the order of 20% false positives and negatives). We are now testing this scheme on Xavier itself. We also intend to replace the current path-planning algorithm with one that develops a *policy*, that is, a specification of which action to take from any location in order to reach the goal optimally. This policy will take into account the probability that pertinent landmarks might be missed, so that, for instance, the robot might prefer to take slightly longer routes (distance-wise) if those routes have a higher probability of success.

For the Lunar rover, local obstacle avoidance is done using a planner adapted from the Unmanned Ground Vehicle (UGV) program [2]. The planner merges terrain maps derived from stereo data, and evaluates the traversability of a discrete number of potential arcs that the rover could follow. Although it has had great success in the UGV program, we have found problems that stem mainly from the inaccuracies in the stereo data and dead-reckoning, which cause the merged maps to be less than reliable. Also, the current planner depends on a detailed model of the vehicle response to predict its traverses over the terrain. If that model is not accurate, the overall system performance suffers. We intend to overcome some of these problems by explicitly representing the uncertainty in the data and vehicle models, and having the planner take that uncertainty into account. For example, we can augment the terrain elevation maps with the variance in the measurement, as predicted by a model of the stereo system. This will give us a model of the confidence we should have in using that data, and hence our confidence in the evaluation of the traversability of an arc. Similarly, we can use uncertainty models of the vehicle performance to predict the likely area that the vehicle will traverse, rather than assuming it will follow a single arc.

Lessons Learned

We have developed the Task Control Architecture, which combines deliberative and reactive control, to help design and implement concurrent distributed robotic systems, and have used the architecture to develop about a dozen robot systems, including a six-legged planetary rover, a Lunar rover, and an indoor autonomous mobile robot. TCA uses task decomposition and task sequencing for its deliberative aspects, and execution monitoring and exception handling for its reactive aspects. The idea is to layer the reactive behaviors onto the nominal plans, thus incrementally increasing system reliability. To further increase reliability, we are currently working on formalizing TCA-based systems, to detect unwanted interactions between behaviors, and are developing probabilistic representations and planning algorithms, to aid in the automated learning of monitors and exception handlers.

What lessons has this experience taught us about designing reliable autonomous robots? First and foremost, reliability does not just "emerge" from a collection of behaviors, even if each behavior is individually fairly reliable. Reliability needs to be structured into the basic system architecture in order to prevent unwanted interactions, resource over-utilization, etc. We have had good success through the hierarchical structuring of systems — task/subtask hierarchies, hierarchies of coarse-to-fine monitoring strategies, and hierarchies of local-to-global exception handling strategies. While this added structuring introduces some run-time overhead, it is nonetheless worthwhile overall because it makes it easier to keep components modular and independent.

Second, we have learned the importance of incremental system development. It is usually impossible to pre-specify everything that could possibly go wrong with a mobile robot that operates in a complex environment. It is a fiction to believe that primitive behaviors can be designed that "do the right thing" in every conceivable situation. The agent architecture must allow for incremental development, and it must facilitate this by enabling components to be added without the need to modify existing components. TCA does this, to a large extent, through its constructs for specifying monitors, exception handlers, and task decomposition strategies. While the original aim in developing TCA was to make it possible to have these components be learned and added automatically, the same philosophy

facilitates building these systems by hand.

Third, we have learned the difficulty of automating the acquisition of such components. The main difficulties are in determining when a problem has occurred and determining what information is relevant to the problem. We need to encode *expectations* about why the robot is performing particular actions. These expectations provide a measure to determine when things are going wrong, and can be used to analyze what must be done to make them right again. Unfortunately, we have found that symbolic, discrete representations alone are insufficient to capture the subtleties of the world needed for making such distinctions. We have therefore been led towards investigating probabilistic, and other, representations of uncertainty. We believe that these representations, in conjunction with more traditional symbolic approaches, will suffice for the tasks of automatically acquiring monitoring and exception handling strategies.

In summary, we have much real-world experience with autonomous mobile robots navigating in uncertain, complex environments. This experience, in turn, has taught us much about the design of software architectures for reliable robot behavior. It has also taught us how far we still have to go to reach the goal of automating the development of reliable mobile robots.

References

[1] R. T. Goodwin. A formal specification of agent properties. Technical Report CMU-CS-93-159, Carnegie Mellon University, Pittsburgh PA. USA, May 1993.

[2] A. Kelly. A partial analysis of the high speed autonomous navigation problem. Technical Report CMU-RI-TR-94-16, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1994.

[3] E. Krotkov, J. Bares, L. Katragadda, R. Simmons, and R. Whittaker. Lunar rover technology demonstrations with dante and ratler. In *Proc. Intl. Symp. Artificial Intelligence, Robotics, and Automation for Space*, Jet Propulsion Laboratory, Pasadena, CA, Oct. 1994.

[4] D. Miller. Execution monitoring for a mobile robot system. In *Proc. SPIE Conference on Intelligent Control*, Cambridge, Massachusetts, 1989. Society of Photo-Optical Instrumentation Engineers.

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

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

[7] R. Simmons. Expectation-based behavior. In *Proc. of International Symposium of Robotics Research*, Hidden Valley, PA, Oct. 1993.

[8] R. Simmons. Becoming increasingly reliable. In *Proc. of 2nd Intl. Conference on Artificial Intelligence Planning Systems*, Chicago, IL, June 1994.

[9] R. Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1), Feb. 1994.

[10] R. Simmons, E. Krotkov, W. Whittaker, et al. Progress towards robotic exploration of extreme terrain. *Journal of Applied Intelligence*, 2:163–180, 1992.

Answers to Questions

Here are my answers to some of the questions posed by the symposium organizers.

Coordination: *Is there a need for central behavior coordination?* Strictly speaking, central coordination is not needed — what is needed is coordination amongst potentially interacting behaviors. This coordination can be either "command arbitration" or "goal arbitration." In command arbitration, coordination occurs right before commands are issued; in goal arbitration, coordination occurs at a higher level, by prioritizing or sequencing subgoals. For most agents, goal arbitration has advantages: 1) it saves resources, since not all behaviors have to be active at all times, and 2) agents can make more informed decisions, since the *purpose* of the goals can be taken into account when determining how to coordinate activities. The latter is harder for command arbitration, since usually, by the time commands are being issued, the purpose is either not accessible or is far removed from the behavior issuing the command.

On the other hand, command arbitration has advantages where the environment is very unpredictable. In such cases, one might want to keep conflicting behaviors active and arbitrate their outcome only at the last moment. Note that goal and command arbitration schemes are not mutually exclusive — architectures can, and should, support both. Also note that neither needs to be centralized, *per se*, since the arbitration mechanisms only need to make pairwise decisions. However, when many interactions can potentially occur, centralized arbitration is often easier to design, implement, and understand what is happening.

Interfaces: *How can human expertise be easily brought into an agent's decisions? How should an agent capture mission intentions or integrate various levels of autonomy or shared control?* A successful architecture needs to enable humans to interact with the agent at any level of abstraction. For example, our work with planetary rovers demonstrates that at times one needs to teleoperate the robot at the joint level, while at other times it is sufficient (and much more efficient) to give heading or positional commands (i.e., go to this X,Y location). Depending on the complexity of the environment and the capabilities of the agent, the human may have to give more or less detailed instructions. The exact form of the interface (textual, spoken, gestures) is not as important as the ability to intervene at any level of hierarchical abstraction.

The hierarchical decomposition of goals (tasks) into subgoals (subtasks) can be used to propagate intentions. This is an extremely powerful method for organizing complex behaviors, and is fundamental to many agent architectures. By reasoning about this hierarchy, an agent can decide such things as when it is making progress towards its goals, when a subgoal is no longer applicable, etc.

Representation: *How should the agent organize and represent its internal knowledge and skills?* In general, agents need three types of knowledge: task decomposition, arbitration rules, and expectations. This knowledge does not have to be explicit — the agent designer can compile it into the architecture — but if it is explicit then the agent will typically be more flexible and understandable to others.

Task decomposition knowledge enables agents to take high-level goals and break them into simpler subgoals. As indicated above, task decomposition is fundamental for achieving complex tasks. If the knowledge is made explicit, agents can combine it in novel ways to achieve new goals in novel situations.

Arbitration rules, also discussed above, are necessary because inevitably there is interaction between parts (behaviors, subtasks) of the agent. Again, these rules can be compiled, statically, into the architecture, but by making them explicit, one can more easily make the behavior of the agent context-dependent, since the agent can reason about which arbitration rules to apply in which situations.

Expectations, which indicate what results subgoals are supposed to have, form the core of getting agents to perform successfully. Expectation knowledge is typically encoded in both execution monitors (what anomalies to look for) and exception handlers (how to get the agent back on track). By making such knowledge explicit, one can learn to recognize and deal with new contingencies. This is extremely important, since most agent failures occur when they encounter situations that their designers did not anticipate. If agents could notice when expectations about their actions were being violated, then they could, at the very least, stop doing stupid things.

Structural: *How should the computational capabilities of an agent be divided, structured, and interconnected? What is the best decomposition/granularity of architectural components?* Obviously, I believe that hierarchy is an important structuring principle. In addition to its use in task decomposition, hierarchy is important in monitoring and exception handling. To conserve resources in execution monitoring, the agent should perceive at the lowest resolution that will distinguish normal from abnormal (expected from unexpected) conditions. Once that determination is made, the agent can then focus its perception to further distinguish the problem. For example, an office navigation robot can use a general "forward progress detection" monitor (perhaps based on wheel encoders). If lack of forward progress is detected, the robot can then determine the exact cause (a blocked corridor, unexpected end of corridor, etc.) in order to determine the appropriate response.

Similarly, exception handling should be hierarchical — the agent starts by applying lower-level exception

handlers, which have more local effects but are applicable in more restricted contexts. If they fail to solve the problem, then the agent should try higher-level, more general strategies. For example, when a blocked corridor is detected, a local strategy is to find a way around the blockage; a more global strategy is find an alternative route; an even higher-level strategy is to give up on the current goal, if no alternative route is available.

These hierarchies should be kept distinct — separate task decomposition, monitoring, and exception handling hierarchies should coexist. The advantage is flexibility and ease of incrementally modifying systems.

Performance: *What types of performance goals and metrics can realistically be used for agents operating in dynamic, uncertain, and even actively hostile environments?* In my mind, the only real performance measurements that make any sense are 1) the range of conditions that the agent can successfully perform its tasks, and 2) the percentage of time it is successful in a given environment. We should act to delineate the space of environments for certain tasks (corridor navigation, office cleanup, etc.). For example, the corridor navigation task can be divided into static/dynamic environments (people, doors opening and closing, etc.), degree of fore-knowledge of the environment (topology, metric information), tightness of fit (e.g., ratio of robot width to corridor width), uniformity of the corridors (same color, same reflectance properties), orthogonal vs. non-orthogonal corridors, etc. Such an analysis of the environment (and task) would go a long way towards providing a basis for comparing the performance of agents.

Simulation: *What, if any, role can advanced simulation technology play in developing and verifying modules and/or systems?* Simulators are invaluable as development tools. They enable the agent designer to test out ideas in a safe environment and to easily run controlled experiments to test the efficacy of different algorithms. Out planetary rover work would be nearly impossible without the use of simulators. We like to build simulators that have the same interfaces as the real robot hardware (including all sensors). In this way, the exact same code that runs on the simulator can be ported, unchanged, to the real robot. This eliminates any translation errors and increases our confidence that the code will work on the actual hardware. The anonymous message-passing communications of our Task Control Architecture makes this easy to do. That said, nothing is ever sure until experiments are done on the robot itself. For all their advances, simulations remain just that — simulations of the real world, and all simulators include the biases of their human designers.

Learning: *How can a given architecture support learning?* Ah yes, learning. The Holy Grail of AI and Robotics. My goal has been to first develop an architecture in which learning can take place, and then to move on to the actual learning aspects. So far, we haven't quite got to the second stage, but the first stage is in reasonably good shape. Mainly, the necessary infrastructure involves delineating the hierarchies needed so that new task decomposition strategies, new monitors, and new exception handlers can be incrementally integrated with minimal effect on existing knowledge. To perform such learning task, the agent needs explicit knowledge of expectations (which is something we are starting to integrate into the architecture). Our efforts at robot learning to date have involved refining parameters of existing knowledge structures, such as learning metric information to augment a topological map, learning new visual landmarks, etc. Work is just beginning on learning more strategic knowledge.