A Real-Time Rover Executive Based On Model-Based Reactive Planning

M. Bernardine Dias
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213, USA
mbdias@ri.cmu.edu

Solange Lemai
LAAS/CNRS
7 ave. du Colonel Roche
31077 Toulouse cedex 4, France
slemai@laas.fr

Nicola Muscettola
NASA Ames Research Center
MS 269-2
Moffett Field, CA 94035, USA
mus@email.arc.nasa.gov

Abstract

This paper reports on the experimental verification of the ability of IDEA (Intelligent Distributed Execution Architecture) effectively operate at multiple levels of abstraction in an autonomous control system. The basic hypothesis of IDEA is that a large control system can be structured as a collection of interacting control agents, each organized around the same fundamental structure. Two IDEA agents, a system-level agent and a mission-level agent, are designed and implemented to autonomously control the K9 rover in real-time. The system is evaluated in the scenario where the rover must acquire images from a specified set of locations. The IDEA agents are responsible for enabling the rover to achieve its goals while monitoring the execution and safety of the rover and recovering from dangerous states when necessary. Experiments carried out both in simulation and on the physical rover, produced highly promising results.

1. Introduction

Robotics space exploration requires autonomous control. While executing critical maneuvers or moving on rugged terrains the speed of the needed control loops does not allow closing the loop with ground control due to large communication delays Limited communication bandwidth and high personnel costs also increase the time and cost for recovering from on-board anomalies if large ground control crews are involved. The need to increase science output and operations safety while reaching for more ambitious and complex exploration goals strongly calls for more autonomous robots.

Some of the most autonomous space systems that have flown [9] or are preparing to fly [4] employ on-board automated planning systems. A planner receives goals either from the ground or from on-board task experts. The planner has access to a declarative model describing the necessary conditions that have to be satisfied in a plan in order to correctly achieve a goal and execute any supporting activities. On the basis of the model, the planner uses a standard planning engine, i.e., a search procedure that efficiently explores a large number of possible ways to concatenate goals and supporting activities. This is done within the temporal and resource constraints intrinsic in the problem. Once a plan has been generated, it is read by a simple interpreter that issues appropriate commands to the performing system and monitors execution feedback returning from it.

Plan driven control is attractive in several respects. Perhaps the most important is the high level of assurance that it can deliver. The declarative model is essentially a constraint-based formal specification of the possible control behaviors of the system. In traditional flight software this specification is typically manually translated into the running code. Plan-based control instead eliminates this error-prone and difficult-to-validate development phase. Provided that the model correctly captures the physics of the devices and the desired control laws, the planning engine will guarantee the correctness of the control software. Of course, this argument relies on achieving a high level of assurance for the search engine. But reuse of the search engine without change across several applications subjects it to several cycles of rigorous testing, intrinsically increasing its reliability. Moreover, engine reuse also make it economically feasible to use high-cost/high-reliability validation such as application of formal methods [6].

However, so far planners are rarely used in on-board control systems for robots. When they are used, the planners are typically relegated to optimizing high-level

task allocation over extended horizon while lower-level control has been achieved with procedural execution [12] or behavior-based control [2]. This situation is partly due to a reaction to early attempts to build plan-based mobile control systems [5] where planning was identified as a principal obstacle to the achievement of reactive behaviors. An important question, therefore, is whether it is possible to build planner-based core controllers that are fast enough to satisfy the reactive requirements of robotic controllers while fulfilling the high-assurance promise of plan-based computation.

This paper describes preliminary work in this direction. We describe the design and implementation of a rover controller that uses planning as the core reasoning engine of a real-time executive. The control system has been demonstrated on the K9 rover testbed (Figure 1) [1] at the NASA Ames Research Center. The tasks performed include some simple mission scenarios requiring the rover to take pictures with the on-board camera and recovering from simple faults such as excessive tilt and roll. The on-board executive was implemented using a general-purpose, planner-based distributed agent architecture, the Intelligent Distributed Execution Architecture (IDEA). It demonstrates IDEA's viability for the implementation of real-time robotic controllers.

This paper is organized as follows. Section 2 gives a brief overview of IDEA agent architecture and describes how planning is integrated at the core of the execution cycle. Section 3 describes the test scenarios run on the K9 rover and how the scenario is modeled by separate IDEA agents. Section 4 reports experimental results while section 5 concludes the paper and discusses future work.

Figure 1 The K9 Rover

2. Structure of IDEA

The most common organizational structure of autonomous control systems that have been used in practical applications is hybrid multi-layered, with several technologically diverse layers cooperating to achieve the robot's desired behavior. In mobile robotics, for example, a common layered controller separates between a low-level functional layer, often organized as a collection of controllers communicating according to a static routing map, and a high level decision layer, typically centered around a procedural execution system [10]. Technological diversity among layers is problematic since each layer's machinery is typically described with a different computational model and supports different programming languages and methods without a clear mapping between them. This is problematic for two reasons. Firstly, it increases the cost and difficulty of building complex autonomous controllers since a roboticist is supposed to thoroughly understand each computational model to be able to effectively program in it. Secondly, it increases the cost of validation and decreases the reliability of the software, since often the same information may need to be represented in two different ways in different layers. Moreover, lack of uniformity between layers increases the difficulty of using automated validation systems.

The Intelligent Distributed Execution Architecture (IDEA) postulates a different approach to the organization of complex autonomous controllers. The basic hypothesis is that a large control system can be structured as a collection of interacting control agents, each organized around the same fundamental structure. Each atomic IDEA agent is structured in the same way and uses a model-based reactive planner as its core engine for reasoning. Each agent is required to operate with real-time guarantees. In fact, each agent has an intrinsic *execution latency*, a time quantum within which all computations needed to execute a "sense/plan/act" cycle must complete, otherwise the IDEA agent is declared faulty and must be taken off-line. The existence of an execution latency allows bridging the perceived gap between AI-based methodologies to control and traditional control theory. In fact, the latency can be directly mapped to a controller's sampling rate, the fundamental measurement of responsiveness in traditional control theory.

Figure 2 describes the core structure of an atomic IDEA agent.

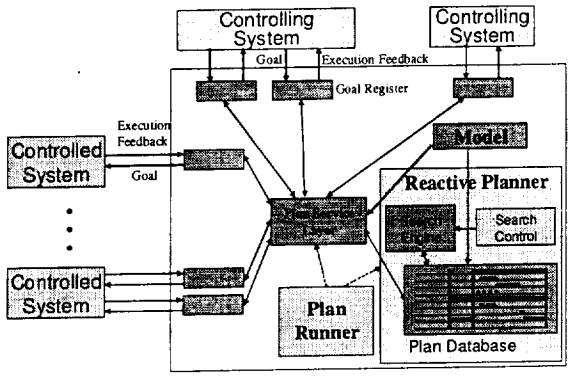

Figure 2 Structure of an IDEA agent

The agent communicates with external systems through a set of *goal registers*. At any point in time a register must contain an active goal describing the "interaction contract" with an external system. The content of the register always takes the form $P(\underline{i} \rightarrow \underline{s})$ where P is the name of a procedure, $\underline{i}$ is a (possibly empty) vector of input values and $\underline{s}$ is a (possibly empty) vector of return status parameters. When the goal is established, all arguments in $\underline{i}$ must be bound to some value $\underline{i_0}$ within the domain of possible values for $\underline{i}$. The contract terminates when either s is bound to a specific value, due to sensory feedback, or a timer associated to $<P, \underline{i_0}>$ expires. The latter allows procedures to be terminated by pre-emption in cases such as lack of response within a maximum allowable wait time. A subsystem interacting with the IDEA agent can be either controlling or controlled. It is controlled if the IDEA agent initially sets the value of the goal register with a new procedure and then waits for the controlled subsystem to set the status s or for the procedure timer to expire. It is controlling in the symmetrical case. A subsystem can be both controlling and controlled by interacting with the IDEA agent with different registers with different communication directions. Subsystems can be other IDEA agents or legacy software and hardware devices whose incoming and outgoing communications can be mapped into a finite set of goal registers maintained by the IDEA agent. The compositionality of the communication infrastructure allows the implementation of arbitrary distributed multi-agent control system structures.

Each goal register must behave according to a "timeline semantic". This means that at any point in time all goal registers must contain an active procedure. This, of course, cannot be satisfied when a procedure returns or must be terminated. In this case the agent goes through an execution cycle whose goal is to eliminate expired or returned procedures from goal registers and replace them with new procedures. The agent must perform this activity with a strict real-time guarantee, within the execution latency associated with the agent. The shorter the execution latency, the faster the IDEA agent can close the control loops in which it is involved.

The module with the responsibility of starting and possibly aborting an execution cycle is the *Plan* Runner. The plan runner can only be activated at discrete times, synchronously with the agent's internal clock. The clock's granularity is the agent's execution latency. If a sensor value is received at time t, this will cause an execution cycle to start at time $k\lambda$ where λ is the agent's latency and $(k-1)\ \lambda \leq t < k\lambda$. Moreover, if the agent decides to start a new procedure during an execution cycle starting at time $k\lambda$, the procedure will be loaded in the goal register at a time τ, where $k\lambda \leq \tau < (k+1)\ \lambda$. Note therefore that in the worst case an IDEA agent's responsiveness, i.e., the maximum temporal distance between a stimulus (sensor value) and its response (the message announcing to the controlled agent that it should start a new procedure), is always 2λ. This permits precise quantification of the reactivity of a control agent, a measure that is usually elusive in control approaches based on planning or other Artificial Intelligence techniques.

The core reasoning in an IDEA agent is performed by the *Reactive Planner*. During an execution cycle, the reactive planner has the responsibility of determining the procedures with which expired goal registers should be loaded. The reactive planner explicitly represents histories for the agent's timelines in a *Plan Database*. These describe both past and future contents of each goal register (either incoming or outgoing) and auxiliary state variables possibly describing non-observable state variables in controlled/controlling systems and internal state maintained by the IDEA agent to implement its control law. In the reference implementation of an IDEA agent, the planner uses a heuristic search procedure implemented through a standard search engine and guided by search control rules implemented in an appropriate search control language associated with the engine. The

planner conducts the search by continuously consulting a *Model*, i.e., a description of how procedures can follow each others on timelines and hence in goal registers. The model also describes in which way start and end of procedures can synchronize in all legal plans (see Section 3.3 for an example). By directly interpreting a declarative model, we believe that an IDEA agent can achieve higher levels of assurance than procedural approaches to plan execution and control.

The IDEA architecture supports several mechanisms for addressing the "planning bottleneck" problem, the problem that has led to the summary dismissal of planning as a core control technology in the past. First of all, note that the architecture assumes the existence of a central plan database for each agent. It is possible for an agent to have several processes, besides the reactive planner, manipulate the plan database. Some of these processes can have the responsibility to build sections of plans over extended periods of time in the future, possibly with the goal of "optimizing" some quality criteria. These processes operate at lower priority than the reactive planner and are controlled by the plan runner through goal registers, i.e., with the same coordination protocol used with external systems. Therefore, as long as the planning horizon over which the deliberative planner is working never intersects the current execution time, deliberative planning can operate in parallel with reactive execution and does not affect the reactivity of the agent. The reactive planner itself may want to operate over planning horizons that are longer than the minimum possible one (one latency interval starting at the current execution time). However, the length of this horizon and the complexity of the model that the reactive planner must use determine the worst case cost for solving a reactive planner problem and therefore determine the agent's latency. Vice versa, if the latency is bound by some characteristics of the controlled subsystems, one can deduce strict limits to the planning horizon as a function of the complexity of the model. Reducing the planning horizon will cause the agent to be more reactively myopic which may require compiling more information in the "control law" timelines in the model or require more extensive deliberative planning in advance (e.g., explicitly representing contingency branches) that allow the reactive planner simply to select an action among those cached in the plan database by the deliberative planner rather than having to synthesize one from scratch every time.

Another way to tune the performance of an IDEA agent is to select a plan database/planning technology with the appropriate expressivity/performance tradeoff. For example, when it is important to reason about time, resources and bound uncertainty, then it could be appropriate to use constraint-based temporal planning technologies such as the one employed in the Remote Agent on-board planner. However, if the model matches an asynchronous discrete event control system, then a propositional representation and fast propositional incremental planning may be better suited to the task and achieve better performance. The IDEA architecture supports the use of different planning technologies by providing a standardized interface, the *Plan Service Layer*, between the planner and the goal register. Different planning technologies can be used as long as they can support a standard set of methods provided by the plan service layer. Also, an appropriate mapping must be defined between the modeling infrastructure of IDEA and the internal modeling needed by different plan database technologies.

In summary, the IDEA architecture provides an implementation of a set of basic services for building agents (goal registers and their input/output communication protocols, the plan runner, the plan service layer, the model) that we believe will be applicable across a wide variety of agents at multiple levels of abstraction in an autonomous control system. The proof of whether this goal can be achieved or not depends both on theoretical analysis and on experimental validations, such as the one reported in this paper.

3. A rover controller using IDEA

We have designed and implemented an IDEA controller for the K9 rover (Figure 1). The K9 rover is a six-wheeled, solar-powered rover complete with a manipulator. K9's mechanisms are a clone of those of the "FIDO" (Field Integrated Design and Operations) rover developed at JPL[11]. The rover's avionics, instrumentation, and its autonomy software were developed at NASA Ames.

The rover carries a variety of instruments on board, including a compass, an inertial measurement unit and

three pairs of monochromatic cameras (WideEye and 2 pairs of HazCams) used for navigation and instrument placement. Other instruments are mounted on an articulated arm that allows their precise placement for contact science. The WideEye stereo pair consists of a stereo pair of CMOS cameras mounted on a 10.93 cm baseline. The individual cameras consist of analog (RS170) output CMOS cameras with a 510x492 pixel resolution. Like the WideEye cameras, the front and rear HazCam stereo pairs consist of stereo pairs of CMOS cameras mounted on a 10.8 cm baseline. The individual cameras consist of analog (RS170) output CMOS cameras with a 510x492 pixel resolution. The rover also carries a pair of high-resolution, color stereo cameras (HawkEye), which consists of a stereo pair of high resolution multi-spectral cameras spaced on a 27.9 cm baseline. The individual cameras utilize a 960x800 CMOS detector with 10 bits/pixel resolution and square pixel format, and the CHAMP, an arm-mounted, focusable microscopic camera developed at the University of Colorado, Boulder. The WideEye and HawkEye camera pairs are fitted on a PanTilt unit.

In this section, we first present the structure of the IDEA controller and its mapping to low-level rover control software. We then describe the test scenario and the models used by each IDEA agent to support this application. The scenario and the models have been tested in simulation and on-board the rover. Some results are discussed at the end of this section.

3.1. Structure of the IDEA controller

Figure 3 depicts the mapping between the IDEA controller and the K9 controllers. The K9 controllers provide a functional layer of capabilities used by the IDEA controller. These capabilities include low-level commands – for instance the simple pan/tilt or camera commands – as well as some more complex behavioral commands, such as "drive to a position". Query functions can be used to obtain sensory information such as the rover's location, pitch/roll/yaw angles and the internal bay's temperature. The overall control software is composed by three subsystems organized in a three-layered hierarchy. The top layer of the hierarchy includes two IDEA agents: the System Level and Mission Level agents. The bottom layer interacts with the System Level agent according to the IDEA inter-agent protocol, although it is not implemented as an IDEA agent. The mapping is obtained through the K9Relay which behaves as a parser/decoder, translating the goals sent by the System Level agent into the corresponding commands or information requests to the K9 controllers. We used CORBA as the underlying messaging infrastructure used to exchange goals and execution feedback between the IDEA agents and to exchange messages between the K9 controllers and the K9Relay.

3.2. Scenario

The IDEA control system has been tested on the following mission scenario. The rover must acquire images from several specified locations. A set of goals is sent to the rover, each consisting of a location and parameters for the camera and the pan/tilt unit. The rover decides in which order to accomplish these goals, monitors their execution and recovers from dangerous states.

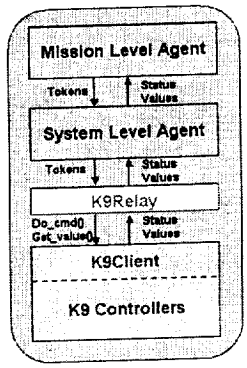

Figure 3 Mapping the IDEA agents to K9

Responsibilities have been assigned to the IDEA agents as follows. The Mission-Level agent receives goals (e.g. from the ground controllers) and decides on their best ordering using a deliberative planner. Execution of the plan at the mission-level sends one goal at a time to the System-Level agent that is responsible for expanding lower-level activities, monitoring execution and planning recovery actions if necessary.

The System-Level agent is responsible for monitoring rover safety while executing its plan. In particular, if safety limits for tilt and/or roll angles are exceeded, the system-level agent immediately stops the nominal execution, orders the rover to backup, executes a turn in place by a set angle, and resume execution of appropriate

actions to achieve the goal. All of this is achieved through local reactive planning and plan execution.

3.3. Model description

The underlying planning technology used in both IDEA controllers is the EUROPA planning technology [7], a direct descendent of the Planner/Scheduler that was part of the Remote Agent [8]. The modeling language used for the agent models is the Domain Description Language (DDL) supported by EUROPA. Thus, designing a model is equivalent to defining a set of parallel timelines, sets of procedure types that can appear on each timeline and a set of constraints for each time interval over which a procedure can extend: temporal constraints between procedure intervals (also called compatibilities), duration constraints and parametric constraints that tie together all token variables (including the interval start time, end time, duration and input and status argument of the procedure).

Search control is implemented through heuristic rules used both by the reactive and deliberative planner. The rules prioritize subgoals that the planner should work on at each step of the search and prioritizes slots on the timelines into which subgoals could be inserted. For the K9 controller, however, only a few heuristics were needed. They were used to prevent the Reactive Planner from trying to bind specific parameters, mainly the parameters corresponding to the output arguments and return status, since their values are determined by the subsystem. Note that in principle it would be possible for the reactive planner to "guess" the return values of procedures. This is particularly important if the planner does look-ahead a few steps in the future or needs to develop contingent plans. In this case, the planner value of the return arguments would be checked with respect to the one actually obtained from the subsystem. If they do not match, then the reactive planner needs to modify the plan according to the value returned from the subsystem which is the true sensor value. Our controller, however, was simple enough that the planner needed only to determine the next action without look-ahead and therefore could afford to leave the value of the return parameters unbound. This behavior is consistent with typical approaches to procedural execution.

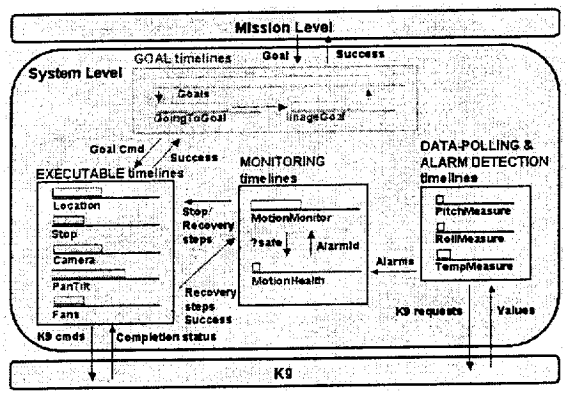

Figure 4 System Level: interactions between timelines

Figure 4 depicts the interactions between the timelines defined in the System-Level's model. There are four types of timelines:

- The *Goal* timelines contain the goal sent by the Mission-Level and manage its completion. One of these timelines is shared with the Mission-Level agent.
- One timeline has been defined for each K9 component controlled by the agent: Location, Camera, Pan/Tilt unit, Fans. These *Executable* timelines contain tokens corresponding to the actual commands sent to the K9 controllers. For each command, a completion status is returned by the K9 controllers.
- To allow the monitoring of the rover safety, one *Data-Polling and Alarm Detection* timeline is defined for each monitored characteristic (pitch/roll angles, temperature, power...). These timelines contain tokens corresponding to information requests to the rover. For instance, at each agent clock tick, a *PitchMeasure* (→ *?alarm ?pitch, ?pitch_rcvd)* goal is sent. The parameter *?pitch* is a status value returning the sensed pitch value, *?pitch-rcvd* an additional status parameter that determines whether the token terminated because a value was received for *?pitch* or because the token was pre-empted, and *?alarm* is another Boolean return status parameter. *?alarm* and *?pitch* are linked by a constraint that sets *?alarm* to True if *?pitch* is greater than a predefined threshold. Once the

Plan Runner has received and posted the value of *?pitch* in the plan database, the Reactive Planner applies the constraint, and a possible alarm is detected.

- For error recovery two other *Monitoring* timelines have been added to manage the different alarms and recovery steps. These timelines are especially useful with regard to the motion of the rover, as different motion alarms can occur at the same time and during the recovery actions. One timeline (MotionHealth) gives the state of the rover at each agent clock tick: if there is an alarm, it identifies what type of alarm it is. Moreover, priorities can be defined between the different alarms. Each alarm corresponds to a specific sequence of recovery steps. The other timeline (MotionMonitor) is useful to manage the next recovery step to execute, depending on the evolution of the state of the rover. By means of compatibilities, the Reactive Planner will then insert the corresponding command tokens on the *Executable* timelines.

Figure 5 gives an illustration of a simpler monitoring with an example of compatibilities for the token *TempReadCompare(→ ?state_fan?temp,?temp_rcvd)* of the timeline TempMeasure. The temperature alarm detection is similar to the pitch case. Once the value of the temperature (output value *?temp*) has been received and posted by the plan runner *(?temp_rcvd* is set to True), the reactive planner applies the following constraints : the parameter-function *new_fan_state()* detects a possible alarm and sets the boolean *?state_fan* to True if necessary, then the compatibility *meets* inserts a command token *DeviceSetFanState(?state_fan→)* on the *Executable* timeline Fans. During the same control cycle a goal is sent to the K9Relay that translates into a direct command to the appropriate K9 low-level controller. This command finally turns the fan on. Note that *DeviceSetFanState* has an empty status vector. This is because we assume that the command will be executed in open loop without direct sensory feedback.

```
(Define_Compatibility
  (SINGLE((Rover_Class TempMeasure_SV))
    ((TempReadCompare(→?state_fan ?temp True))))
:duration_bounds [*temp_freq* *temp_freq*]
:parameter_functions
  (new_fan_state(*tempthreshold* ?temp ?state_fan))
:compatibility_spec
  (AND
    (meets (SINGLE ((Rover_Class Fans_SV))
          ((DeviceSetFanState (?state_fan→)))))))
```

Figure 5 Example of compatibility for the token TempReadCompare

The system-level model contains only forward chaining compatibilities, since it is designed for a purely reactive agent, planning over an horizon covering only one execution latency ahead in reaction to new sensory information or new goals.

As stated before, the Mission Level agent receives a set of goals from the ground controllers. It uses deliberative planning to find the best ordering of the goals and sends one goal at a time to the System Level agent for expansion and execution. The Mission Level monitors the completion of each goal and can replan if necessary

The underlying model contains three types of timelines. A set of *Internal* timelines is used by the deliberative planner to find the ordering of the goals. Deliberative planning is managed by means of a specific *Planner* timeline that contains Planning tokens which parameters specify, notably, the start and end times of the planning horizon. The execution of such a token triggers the corresponding planning process. Finally, the plan resulting from deliberative planning (i.e. a sequence of goals) is put on a *Goal* timeline. This timeline is shared between the two agents. Its execution by the Reactive Planner at the Mission Level communicates one goal at a time to the System Level and monitors the completion status returned back.

The System Level has been tested on board the K9 rover (with one goal sent by the Mission Level from a distant machine). Deliberative planning and interaction between the two agents have been tested in simulation.

4. Results

During the tests on board, the rover has successfully accomplished its goal while correctly responding to