Model-Unified Planning and Execution for Distributed Autonomous System Control

Pascal Aschwanden[4], Vijay Baskaran[4], Sara Bernardini[6], Chuck Fry[4], Maria Moreno[5], Nicola Muscettola[1], Chris Plaunt[3], David Rijsman[2], Paul Tompkins[4]

[1]Lockheed Martin Advanced Technology Center, Palo Alto, CA 94304
[2]Mission Critical Technologies, NASA Ames Research Center, Moffett Field, CA 94035
[3]National Aeronautics and Space Administration, NASA Ames Research Center, Moffett Field, CA 94035
[4]QSS Group Inc., NASA Ames Research Center, Moffett Field, CA 94035
[5]Universidad de Alcala, Madrid, Spain
[6]Universitá di Trento, Trento, Italy

{pascal, bvk, sbernard, cfry, malola, cplaunt, rijsman, pauldt}@email.arc.nasa.gov, nicola.muscettola@lmco.com

Abstract

The Intelligent Distributed Execution Architecture (IDEA) is a real-time architecture that exploits artificial intelligence planning as the core reasoning engine for interacting autonomous agents. Rather than enforcing separate deliberation and execution layers, IDEA unifies them under a single planning technology. Deliberative and reactive planners reason about and act according to a single representation of the past, present and future domain state. The state obeys the rules dictated by a declarative model of all relevant aspects of the domain - the subsystem to be controlled, internal control processes of the IDEA agent, and interactions with other agents. We present IDEA concepts – modeling, the IDEA core architecture, the unification of deliberation and reaction under planning – and illustrate its use in a simple example. Finally, we present several real-world applications of IDEA, and compare IDEA to other high-level control approaches.

Introduction

Autonomous control systems classically subdivide into three layers – a functional layer that provides both tight, fast control of low level system hardware and raw sensor data; a deliberative layer that enables long-range planning and scheduling of control actions using a model of system behavior in the environment; and a reactive layer that uses sensor, timing and event data to synchronize the enactment of plans with the often unpredictable timing and behaviors of the actual controlled system and nature.

One complaint of this architecture is that the planning and reactive layers are only tenuously connected through transmission of a plan – they may use models with disparate levels of fidelity, plan execution logic may differ from the logic that was used to generate the plan, and plan translations from the planner representation to that used by the executive may necessitate over-simplification or mis-representation.

The Intelligent Distributed Execution Architecture (IDEA) is a real-time architecture that exploits artificial intelligence planning as the core reasoning engine for interacting autonomous agents. Rather than enforcing separate deliberation and execution layers, IDEA unifies them under a single planning technology that operates on a single representation of the past, present and future state. Furthermore, IDEA unifies the entire domain description – the descriptions of the controlled subsystem and internal control logic, the coordination of control layers (e.g. how to interleave plan execution and deliberation) and interactions with other agents - in a single model. Because it unifies deliberation and reaction, IDEA enables consistency, coordination, flexibility and sophistication that is not possible in classical, three-layer approaches.

IDEA is also unified under the tenets of real-time control. Reactions to events are synchronized to a global clock, and must complete within the minimum time quantum for the application.

We refer to IDEA controllers as *agents*. Since modeling is so central to IDEA agents, we begin by describing IDEA models and the IDEA modeling language. We then describe the thin layer of functions and services available in all IDEA agents. Planners are specialized IDEA components. We continue by discussing how IDEA adapts general AI planning, via the model, to reaction and deliberation for specific domains, and how the Reactive Planner orchestrates the operation of an agent. We conclude with some examples of IDEA agents fielded in complex control tasks, and a brief comparison of IDEA with other technologies.

Domain and Agent Modeling

At the heart of each IDEA agent is a declarative model describing the system under control of the agent and the interactions with other agents, collectively known as the domain.

IDEA Domain Descriptions

IDEA models are represented in XIDDL (XML IDEA Domain Definition Language), an XML encoding of a generic domain description language for simultaneous constraint-based planning and execution.

An XIDDL domain description consists of three major pieces. First, there is a domain model. In short, the domain model describes the union of all feasible states of the domain. It consists of a declaration of:

- **Objects:** (e.g., rovers, heaters, cameras, UAVs, targets, planners, paths). The whole domain is built up from compositions of objects about which the planners will plan, and with which the agent will interact.
- **Attributes** (or "timelines"): Each object has one or more "timelines" which declare the aspects of that object of interest (e.g., a communication subsystem object might have attributes describing its current configuration, current activity, power state and so on). Some such attributes are used internally, some are controlled by external agents, and some are used to control subsystems.
- **Predicates** (or "tokens"): Attributes of each object can only be in one mutually exclusive, parameterized state at any given time (e.g., a domain might require that a path planning subsystem must either be "idle", or that it must be "planning" for a particular request). Predicates describe these states. Each such predicate consists of a name and zero or more parameters of declared types. Parameters may be controlled internally (by a planner), or externally (by exogenous events).
- **Compatibilities:** The constraint "rules" by which the parallel, mutually exclusive attribute state transitions are controlled and monitored. Compatibilities describe the temporal relationships between predicates, as well as the relationships between predicate parameters, that are feasible in the domain. Once a predicate exists, compatibilities define the subset of states and parameters that could have existed in the past or might exist in the future on the same attribute; or that existed in the past, exist in the present and might exist in the future on other attributes. Hence, compatibilities describe predicate transitions and enforce feasible behavior. Special predicate parameters represent the different transition configurations that can occur from a given predicate, as given by the compatibilities. Therefore, either planners or exogenous events can generate particular transitions by setting the value of the parameters.

Via these mechanisms, domain models enable a rich description of natural and controller behaviors and interactions during execution. From the perspective of an IDEA user, the purpose of the domain model is to "program" the agent to respond appropriately in all feasible situations.

The second element of the domain description is the agent topology and configuration description, consisting of the "latency" (the longest acceptable time for the agent to provide a reactive response to a situation), a declaration of all of the known initial subsystems and their initial state, the communications "channels" between the agents and subsystems in the domain, and finally, a declaration of the agent's internal processes.

The third piece of the domain description is a declaration of the planners in the agent. This declaration controls how the various planners interact with each other and with the plan database. In particular, this declaration includes the planner scope, e.g. what attributes and predicates each planner can see, the horizon over which each will plan, the heuristics (i.e., search strategy) each will use, and finally, several domain- and planner-specific recoveries and the situations in which they apply (e.g. relaxing the future of one or more timelines, relaxing the whole database and loading a "standby state", and so on).

Language Services for Modelers

Though the model, agent and planner declarations are technically distinct, XIDDL represents them in a consistent manner which allows us to leverage standard XML tools to provide powerful "compile time" cross-validation of the contents of these elements (e.g., all states declared in the initial state must be legal with respect the domain model). We also leverage this generic domain description to provide support, via translation, for multiple "back end" modeling languages.

Currently, IDEA supports both Domain Description Language (DDL) and New Domain Description Language (NDDL) back end languages; both generic and back end language-specific "semantic checking" (e.g. legal constant names, unique identifiers, missing, unused or conflicting definitions, and so on); and automatically generated model documentation, including both embedded "documentation strings" supplied by the modeler and a navigable representation of the internal relationships between objects, attributes, predicates, guards, parameters, and conditional subgoals.

The Satellite Domain: A Simple Example

In this section, we introduce a simplified version of a real application concerning the operation of a satellite. Throughout the paper, we will use this case as a running example.

Consider a satellite which orbits the Earth in order to take pictures of some interesting targets. The satellite has a number of instruments on board and each of them has several operating modes. A camera instrument can take a picture of an object only if it has been previously calibrated and if the satellite points in the direction of the object. The satellite has power to operate only one instrument at a time. The tasks performed by the satellite and by an instrument can succeed or fail; each must generate proper messages to acknowledge its results.

Table 1: Defining Attributes of a Class in XIDDL

```
<define_object_class type="dynamic">
   <name>Instrument_Class</name>
   <attr>Satellite_SV</attr>
   <attr>Mode_SV</attr>
   <attr>ActionInst_SV</attr>
</define_object_class>
<define_object_class type="dynamic">
   <name>Satellite_Class</name>
   <attr>ActionSat_SV</attr>
   <attr>PowerStatus_SV</attr>
</define_object_class>
```

We intend to analyze how an IDEA agent represents and controls this system, starting from the XIDDL model that describes it. The domain has two main classes of objects, one representing a generic instrument and the other a generic satellite. For the satellite, we are interested in two aspects: which task it is performing and to which instrument it is giving power. So, we define two attribute timelines, one for each of these aspects (see Table 1 for the corresponding XIDDL). For the instrument class, we want to represent which operation it is executing, which mode it is using and on which satellite it is located. We define three attributes corresponding to those pieces of information. Again, see Table 1 for the XIDDL specification.

Table 2: Defining Predicates of an Attribute in XIDDL

```
<define_member_values>
   <object>
      <class>Instrument_Class</class>
      <attr>ActionInst_SV</attr>
   </object>
   <pred>Calibrate</pred>
   <pred>TakeImage</pred>
   <pred>Idle</pred>
</define_member_values>
<define_member_values>
   <object>
      <class>Satellite_Class</class>
      <attr>ActionSat_SV</attr>
   </object>
   <pred>TurnTo</pred>
   <pred>Pointing</pred>
</define_member_values>
```

In our model, an instrument can be in one of three mutually exclusive states: it can be calibrating, taking a picture or idle (in the case when it is not doing either of the other two actions). So, we define three predicates for the "ActionInst_SV" timeline which represents the possible procedures of an instrument as shown in Table 2. Turning our attention to the tasks performed by a satellite, we define two activities for the "ActionSat_SV" timeline: one describes the action of turning to a new direction and one specifies the action of pointing at a target (also in Table 2).

Table 3: Defining Parameters of a Predicate in XIDDL

```
<define_procedure>
   <name>TurnTo</name>
   <call_args>
      <arg>
         <type>DirectionName_Class</type>
         <name>turnDir</name>
      </arg>
   </call_args>
   <return_status>
      <type>ReturnResult</type>
      <name>resultTurn</name>
      <flag>flagTurn</flag>
   </return_status>
</define_procedure>
```

Table 4: Defining Compatibilities in XIDDL

```
<define_compatibility>
   <master>
      <class>Satellite_Class</class>
      <attr>ActionSat_SV</attr>
      <pred>TurnTo</pred>
   </master>
[...some model omitted...]
   <subgoals>
      <or>
         <arg>resultTurn</arg>
         <case>
            <value>OK</value>
            <meets type="single">
               <class>?_object_</class>
               <attr>ActionSat_SV</attr>
               <pred>Pointing</pred>
               <constraint name="eq">
                  <arg>pntDir</arg>
                  <master>turnDir</master>
               </constraint>
            </meets>
         </case>
         <case>
            <value>Failed</value>
            <meets>
               <class>Satellite_Class</class>
               <attr>ActionSat_SV</attr>
               <pred>TurnTo</pred>
            </meets>
         </case>
      </or>
   </subgoals>
</define_compatibility>
```

Now consider how IDEA describes a predicate, taking "TurnTo" as an example. We want to model this procedure in such a way that the IDEA agent can send the satellite the new pointing target and receive a message from the satellite representing the feedback of the action, that is, if it succeeded or failed. To this end, we use two parameters, a <call_args> "turnDir" to represent the information flowing from the agent to the external system (the satellite, in this case) and a <return_status> "resultTurn" for the information flowing in the opposite direction, from the satellite to the agent. Table 3 shows the XIDDL.

The last part of the model describes the constraints that predicates and attributes have to satisfy. Considering again the "TurnTo" predicate as an example, we want to define the following behavior: if the turning action succeeds, then the satellite will switch into the "Pointing" state, where the pointing direction is target of the "TurnTo". Instead, if the turning action fails, the satellite has to try again to turn to the proper position. See Table 4 for the XIDDL.

The specification of the other compatibilities follows this pattern and expresses all the temporal and causal relationships between predicates in the same attributes and between them.

The Core IDEA Architecture

All IDEA agents provide a thin layer of core services for model management and data storage, and event and data transport and response. Figure 1 illustrates the architecture of an IDEA agent.

Model Management and Data Storage

As a basis for decision making, IDEA agents maintain a database, the Plan Service Layer (PSL) database (see Figure 1), which stores and updates the past, present and future state of objects in the domain. The PSL triggers state changes for specific object attributes in reaction to events that have occurred, updates to parameter values, or the passage of time. It propagates the effects to other objects, attributes, predicates or state parameters, as necessary, according to the model.

The PSL provides consistency checking to verify whether the state of the agent continues to conform to the model compatibilities. The PSL also provides data manipulators that create new instances of object classes and new predicates on attributes, assign particular values to variables in the state, and retract assignments to variables. The PSL design does not commit itself to a specific technology to provide these services. However, IDEA currently utilizes the EUROPA [Jonsson et al. 2004] constraint propagation package as the basis for one of the PSL instantiations.

Event and Data Transport and Response

Internal and external events and data are communicated as IDEA messages. IDEA defines several message types symbolizing different types of events and data transmission:

- **Start Message:** Signals the start of a predicate state on a particular attribute (timeline).
- **Stop Message:** Signals the termination of a predicate state on a particular attribute.
- **Status Message:** Signals the outcome of a particular predicate (e.g. the success or failure of a commanded activity).
- **Value Message:** Signals the transmission of data (e.g. readings from a sensor or parameters from another agent) associated with a particular predicate.

The *Agent Relay* (see Figure 1) is responsible for relaying incoming and outgoing messages within an IDEA agent; it relays outgoing messages to *Communication Relays* and incoming messages to *Reactors*, both described below.

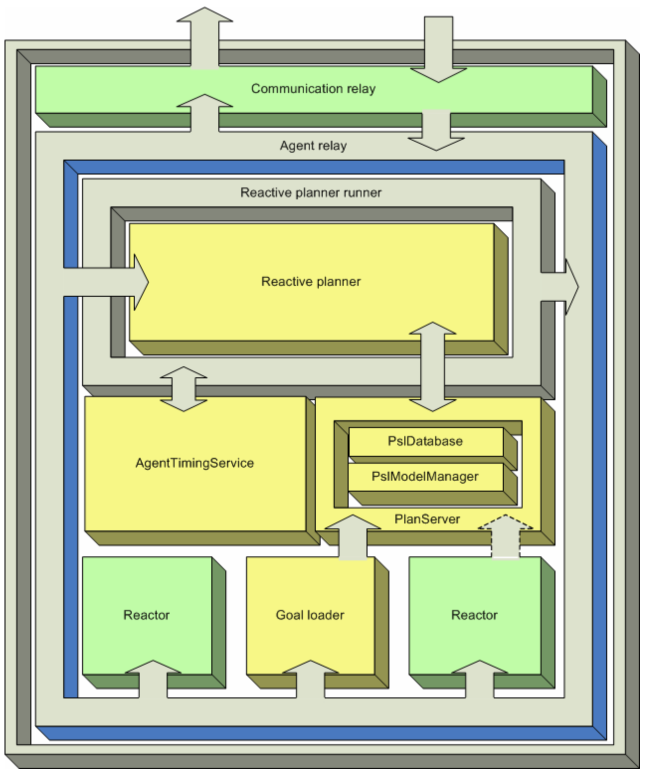

Figure 1: Architecture of an IDEA Agent

Communication Relays (the box at the top of Figure 1) transmit data and events to other agents or systems. In the agent configuration in the model, an IDEA user can associate a Communication Relay with any attribute type or specific attribute instance to enable it to send messages to external systems. As events occur on the attribute (e.g. the start of a new predicate or the status on its completion), the Agent Relay forwards corresponding messages to the associated Communication Relay. Communication Relays typically do not interpret the content of the message, but limit themselves to translating (marshalling) data to the representation of the underlying communication technology.

IDEA provides a base Communication Relay class that is specialized for different communication protocols, including for Carnegie Mellon University's Inter-Process Communication (IPC) and CORBA. These can be further specialized to enable translations into specific external system conventions. For example, two IDEA agents can use the built-in IPC Communications Relay to transmit events, data and status. However, to communicate with a non-IDEA system (e.g. a spacecraft avionics processor, or a robot functional layer), an agent would require a specialized Communications Relay to convert IDEA messages into commands or remote function calls understood by the external system.

Reactors, unlike Communications Relays, interpret the content of messages and react according to the behavior specified in the Reactor source code (see their arrangement at the bottom of Figure 1). By associating a Reactor with a specific predicate or predicates in the domain model, an IDEA user can cause it activate under specific circumstances defined in the compatibilities. A Reactor is either associated with a predicate type on all instances of an object class or on a specific object. Messages representing events for that predicate type are passed by the Agent Relay to the affiliated Reactor. Messages for tokens with no explicit token affiliation are sent by the Agent Relay to a user-specified default Reactor, typically the Reactive Planner, described later.

The IDEA architecture provides a base Reactor class from which applications can define custom Reactor modules, with custom behaviors, including full access to the PSL. Examples of Reactors include reactive planners, deliberative planners, goal loaders and agent shutdown reactors.

Model-Unified Planning and Execution

IDEA unifies planning and execution by allowing multiple planners, operating over different temporal horizons, to reason and act on a single representation of the domain state history. Each planner is a Reactor, with access to the PSL, whose planning horizon, scope over the database, and priority with respect to other Reactors can be customized according to the needs of the application.

At minimum, IDEA agents call on a Reactive Planner (the default Reactor) whose priority is higher than all other Reactors and whose planning horizon is typically the minimum time quantization for the application. Optionally, IDEA agents can include other, lower priority Reactors, including deliberative planners and goal loaders that typically plan over longer horizons.

In the next section we briefly introduce the EUROPA framework, the basis technology for all built-in IDEA planners. The following sections then discuss the specifics of the Reactive Planner and two long-horizon planners.

EUROPA

EUROPA [Jonsson et al., 2004] is a framework for representing and solving constraint satisfaction problems (CSPs), with an emphasis on temporal constraint networks. EUROPA is descended from HSTS [Muscettola et al. 1997] and implemented in standard C++.

All built-in IDEA planners use a straightforward chronological backtracking search engine implemented in EUROPA. The same algorithm is used for both deliberative (long horizon) and reactive (near-term) planning.

Under constraint satisfaction, all domain state parameters (e.g. predicate start times, end times, durations, and additional typed parameters) are represented as sets of values. Numerical parameters (integer and real types) are represented as intervals. A parameter can freely take any value contained in the set or interval. A key EUROPA concept is that constraint propagation, as performed when adding new information to the domain state or during search, serves only to further restrict these sets. In the context of planning and execution, constraint propagation is the method by which time, external events and/or a planner (through decisions) narrows the set of possible choices for parameters (including those representing start times, end times, etc). When a parameter set collapses to a singleton value, it is the only legal value under the given circumstances.

Reactive Planner

The Reactive Planner is the conduit through which an IDEA agent responds to internally and externally-generated events, data and the passage of time. As the default Reactor, the Reactive Planner receives all incoming messages not explicitly modeled to be associated with another Reactor. For each message it receives, the Reactive Planner updates the state of the PSL with the message content. Using the model compatibilities (constraints), it propagates the effects of the update to other predicates or parameters. Further restricting the values of predicate parameters may cause the completion or termination of some events or activities and the start of others.

A principal role of the Reactive Planner is to maintain continuous agent execution – that is, to prevent gaps between predicates during state transitions. The end of a predicate can be signaled by a message or can be inferred with the passage of time according to the temporal constraints imposed by the model. The Reactive Planner makes sure that each attribute has a feasible next predicate whenever the current predicate ends. It must solve a constraint satisfaction problem (planning) to find the successor states that collectively conform to the domain model. The horizon of this planning must be extremely short to ensure the operation completes within the minimum time quantum (see Agent Latency below) of the agent – going beyond it could leave a gap between predicates in some scenarios. Once it has determined the start of a new state or activity, the Reactive Planner must signal the event via a message that communicates it to the external world.

As time progresses and messages arrive, the Reactive Planner will be invoked over and over again to execute its responsibilities.

Reactive Planning in the Satellite Domain

Consider again our running example concerning the operation of a satellite. Suppose, for instance, that a "TurnTo" predicate is currently under execution. Once the activity ends, the Reactive Planner is in charge of deciding which token has to be activated next. Just before the termination of the activity, the satellite returns a Status Message indicating whether the result is “OK” or “Failed”. The Reactive Planner consults the subgoals for the “TurnTo” predicate (see Table 4) in the model. If the “TurnTo” fails, the Reactive Planner applies the corresponding case and will activate another “TurnTo” predicate; otherwise it will activate a “Pointing” predicate.

Deliberative Planners and Goal Loaders

IDEA agents may also utilize traditional, deliberative planners to create plans that achieve a set of goals, to elaborate on existing plans or to load externally-generated plans that specify events far into the future. As with the Reactive Planner, custom planners are IDEA Reactors. They can differ from the Reactive Planner in several important ways:

- **Horizon:** Custom planners are often configured to reason deliberatively by extending their planning horizons to many minimum-time quanta, thereby giving them greater influence over future events.
- **Scope:** The scope of attributes over which custom planners can reason and operate is adjustable. This enables both general planners that create plans over the entire domain and focused planners that reason about specific aspects of the domain.
- **Priority:** Deliberation often requires more computation time than the duration of a reactive cycle. In order to maintain reactivity to internal and external events during deliberation, custom planners must be given lower priority with respect to the Reactive Planner. This allows the Reactive Planner to curtail deliberation if events force a timely response.
- **Configuration:** If desired, a user can custom-configure the search heuristics for each custom planner, causing potentially very different search behaviors.

Custom planners are associated with specific predicate types in the agent configuration. By adding horizon and scope parameters to these predicates, IDEA users can enable an agent to automatically adapt those characteristics to the domain state. The invocation of a custom planner happens in same way as the initiation of any other state or activity – through the action of the Reactive Planner in starting the planner’s associated predicate.

Deliberative planners in IDEA have long horizons and solve for new plans or elaborate existing ones. Goals might be specified to the agent in the initial state specification, or by some other mechanism. Through search, a deliberative planner must find a plan that links the current domain state to the goals. At each search step, the planner makes a decision (e.g. adds a new predicate to an attribute, picks a value for a parameter, etc) and commits it to the PSL. Occasionally the planner must retract decisions to explore other parts of the plan space. Because the process minimally restricts temporal parameters, resulting plans are typically temporally flexible, that is, the predicate start and end times and durations are defined over time intervals. During plan execution, and subject to these intervals, it becomes the job of the Reactive Planner to determine precisely when a predicate starts and ends. IDEA’s built-in deliberative planner allows the Reactive Planner to interrupt between each decision. In this way, an agent can continue to react to events even as protracted deliberation continues.

The goal loader is designed to import externally-defined goals (a temporally flexible plan) into the PSL. The goal loader parses an application-dependent goal representation into an intermediate data structure, and then invokes the built-in deliberative planner to populate the PSL with the new data. Because the plan is largely or fully specified and presumably adheres to the model, the search is far more straightforward than for full plan generation.

Deliberative Planning in the Satellite Domain

Resuming the satellite example, if the goal is to take a picture of a target in an attitude different from the initial attitude, a planner could formulate a plan where the satellite first turns to the new attitude and, once the satellite is pointing in the correct direction, the camera calibration occurs and the camera takes a picture. The start times, end times and durations of the predicates in such a plan might be flexible; this would give the Reactive Planner the freedom to activate and deactivate predicates at the appropriate times, based on external events.

IDEA in Operation

Having introduced the notion of model-unified planning and execution, we now describe how the Reactive Planner orchestrates the operation of an IDEA agent.

Reactive Cycle

In response to events, the Reactive Planner undertakes a number of activities to update the PSL database time, events and data, to maintain PSL database consistency, to end predicates (according to events and the model) and start new ones to ensure continuous execution, and, if necessary to recover from database inconsistencies or timeouts. This series of activities is the *reactive cycle*. Table 5 lists all the steps of the reactive cycle.

The entire reactive cycle must fit within the characteristic time of the agent, called the *latency*. The reactive cycle contains three update steps (2, 4 and 7), three corresponding optional recovery steps (3, 5 and 8) and a communication step (9). The operations in the update steps were introduced in the Reactive Planner section above. The following sections describe the agent latency, the conditions under which recoveries are necessary, and steps that an agent can take to recover.

Table 5: The Reactive Cycle

Step	Description
1	Lock the database such that only the Reactive Planner has access.
2	Update time in the database according to the clock.
3	If Step 2 causes a database inconsistency, recover using the recovery registered for this type of inconsistency.
4	Handle incoming messages, and update the database accordingly.
5	If Step 4 causes a database inconsistency, recover using the recovery registered for this type of inconsistency.
6	End predicates if they are controllable and have a successor (and additional logic)
7	Execute a chronological backtracking search to determine: • the value of variables in the scope of the Reactive Planner • the next predicate on each remaining attribute
8	If Step 7 created an inconsistency, or no plan could be found, recover using the recovery registered for this type of inconsistency.
9	Communicate newly started executable predicates and the values and status corresponding to goal tokens that just ended.
10	Unlock the database such that other planners may gain access.

Agent Latency

IDEA agents are synchronized to a clock provided by the Agent Timing Service. The *latency* (λ) of an execution agent is the upper bound on the duration of a reactive cycle. It is also the smallest time unit (sampling rate) with which an IDEA agent operates. However, as will be described later, we have also implemented experimental agents that operate at a sampling resolution that is smaller than one latency; such an agent is capable of responding to externally triggered events with a sub-latency granularity.

During execution, the Reactive Planner process sleeps until events prompt it to respond. When an external system posts an asynchronous event (such as a status message), or an intrinsic state transition is required as per the plan being executed, the Reactive Planner Runner schedules a wakeup with the Agent Timing Service. The wakeup time marks the beginning of a reactive cycle, as listed in Table 5.

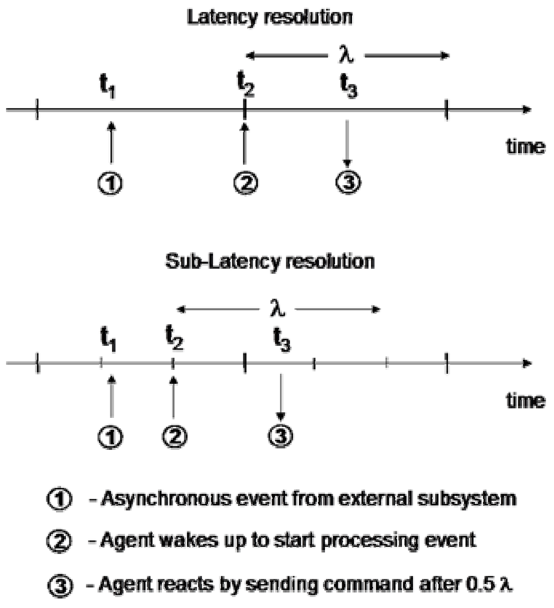
Latency resolution
λ
t_1
t_2
t_3
time
①
②
③
Sub-Latency resolution
λ
t_1
t_2
t_3
time
①
②
③
① - Asynchronous event from external subsystem
② - Agent wakes up to start processing event
③ - Agent reacts by sending command after 0.5 λ

Figure 2: Agent Reaction Times at Full- and Sub-Latency Resolutions

The significance of latency (and sub-latency) is illustrated in Figure 2. Let us consider a simple scenario where an asynchronous event (labeled **(1)** in the figure) is posted by an external system at time t_1. Let us also assume that the domain model is written such that the IDEA agent is required to post a command (labeled as **(3)**) in response to the asynchronous event. As soon as the asynchronous event is sensed by the agent, a wakeup is scheduled by the Planner Runner (labeled **(2)**) for time t_2. Let us also assume that it takes the agent λ/2 time units to process and issue the response command. In the case of an agent with a resolution equal to λ, the wakeup time is scheduled for the next latency grid value. For a sub-latency agent, it is set for the next sub-latency grid value. The agent wakes up at the scheduled time in both cases and posts a response after half a latency. It can be shown that the worst case response time is the same for both agents (2 λ) but on an average, the sub-latency agent has a better response time.

Timeouts and Recovery

If the Reactive Planner is unable to complete its reactive cycle within one latency, the agent flags a "controller timeout" error. The IDEA framework provides the user with modeling hooks to define an appropriate course of action to gracefully terminate in the event of a controller timeout error. One commonly used approach is for the agent to revert to a default set of standby states and then take the necessary recovery action.

Inconsistencies and Recovery

During execution, events may occasionally be in conflict with the past, current or future (predicted or planned) state represented in the PSL database. The plan, expressed as the future state, is only a specific instance of all possible scenarios that might happen in practice. If a model does not properly predict some event, or, under incomplete information, if the plan incorrectly assumes the particular event will not occur, the occurrence of the event will cause an inconsistency in the database. Events that cause inconsistencies include receiving an unexpected response from the subsystem being controlled, receiving a response at an unexpected time, or not receiving one at all.

When such unforeseen events take place during execution, the Reactive Planner, with the assistance of a deliberative planner, must either be able to make the plan consistent, or discard it entirely and generate a new one. The Reactive Planner initiates plan repair by freezing (preventing removal of) all past state known to be true and removing certain constraints on present and future plan state. Then, using a repair strategy encoded in the agent model, a deliberative planner can take appropriate steps to restore consistency. The actions of the Reactive Planner are often enough to enable a deliberative planner to restore consistency without explicit recovery actions. In other situations, specific subsystem fault recovery steps might be needed to resolve the conflict that triggered the fault before continuing with the normal course of operation. In still other cases, repair actions might discard the current plan that is being executed and replace it with one that the controlled subsystem can handle in its degraded capacity.

Recovery is more complicated when an inconsistency occurs during a reactive cycle between steps of deliberative planning or goal loading. Recall that the built-in IDEA deliberative planner cedes control of the PSL database between each decision. Any event will cause the Reactive Planner to interrupt the deliberative planner, to insert the effects of the event into the PSL, and to check for consistency. If the new events come in conflict with the existing partial plan, the Reactive Planner undoes the decisions of the deliberative planner or goal loader, assigns a failed status to the execution of the planning or goal loading predicate, and assigns a new start to the affected attributes.

Applications

IDEA agents have acted as high-level controllers for a number of space- and military-relevant systems. We summarize a few in the following sections.

K9

The Collaborative Decision Systems (CDS) project demonstrated a wide range of technologies applicable to human-robot exploration missions to the surface of the moon or Mars [Pedersen et al. 2006]. In this context, the K9 robot performed autonomous surveys of multiple pre-designated rocks, and relayed data back to “astronauts” in a nearby habitat. The K9 Plan Manager and K9 Executive were both IDEA agents.

The K9 Plan Manager received goal targets (rocks) from the “astronauts”, and created plans in two stages. First, it used a map of known terrain and the locations of target tracking acquisition points and goal positions to create a minimum-distance network of paths. Then, using a EUROPA-based activity planner, considered the path network in creating a concurrent activity plan that achieved as many of the science measurement goals as possible within the allotted time. Plans encompassed navigation, target acquisition and tracking, instrument placement and image sampling activities. The Plan Manager sent those plans to the K9 Executive.

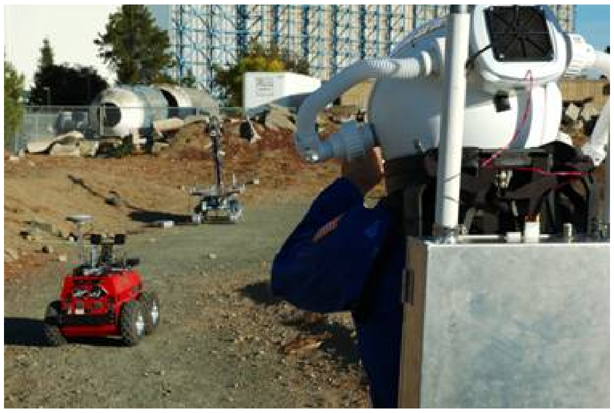

Figure 3: Gromit (left) and K9 (center) with an “Astronaut” in the NASA Ames Marscape

Sharing a large segment of the domain model with the Plan Manager, the Executive loaded the plans into its own PSL database, and executed the plans by dispatching commands to the K9 functional layer and monitoring progress from the system. The Executive enabled fault responses for some commands, and also relayed telemetry and plan execution status back to the Plan Manager. Via the Plan Manager and the Executive, “astronauts” could also manually terminate plan execution to re-specify targets on the fly.

Gromit

Gromit was also part of the CDS project at NASA Ames. Its main role was to pre-survey rocky areas for targets worthy of later investigation by K9 or astronauts. Gromit pursued a list of position way-points, collected stereo panoramic images of specific areas, and built 3D models of the areas covered in the images. Gromit could also respond to voice commands from space-suited astronauts to go to specific locations to collect survey data. As with the K9 rover, Gromit implemented an IDEA Executive agent. The Gromit Executive called on two focused deliberative planners – one for image-taking and another for mobility.

It received high-level goals from astronauts and both the deliberative planners acted on them to produce plans that interleaved mobility and image-taking. Importantly, unlike the K9 Executive, the Gromit Executive coordinated the robot's individual functional layer modules.

The Gromit model decomposes into five distinct parts: an interface to enable immediate commanding by astronauts; an interface to planned activities (via a transmitted plan); a core logic of the high-level executive that allows it to de-conflict astronaut voice commands from planned actions, and to map high-level commands (astronaut and plan) into Gromit functional layer primitives; and the Gromit functional layer itself.

HURT

The HURT (Heterogeneous Urban RSTA Team) project implemented a mixed-initiative (combining human and autonomous decision makers) environment consisting of a manned command center, humans on the ground, and several unmanned autonomous aircraft working in coordination for Reconnaissance, Surveillance and Target Acquisition (RSTA). In this project, IDEA was used as a component of the Planning and Execution system (PLEX). In this role, the PLEX agent received dynamic notifications about what resources it had available to plan with (i.e., which aircraft, and the capabilities of each) and a series of task requests from the command center for coverage and tracking of various dynamic targets. The PLEX agent's job was to coordinate these tasks with respect to the available aircraft and their capabilities; motion, geographic location of both of the aircraft and targets; temporal constraints; and task priorities.

In this domain, the PLEX Plan Manager generated a plan based on the available information, and commanded the aircraft to service all of the tasks for which sufficient and appropriate resources were available, in priority order. In particular, not all aircraft were appropriate for all types of requests. Hence, the planner had to reason with respect to several types of resource constraints. When a new high priory task arrived, the Plan Manager generated a new plan that took this change into account.

A recent version of the PLEX Agent was tested in the field with four live aircraft. The most recent version, running at a latency of 0.2 seconds, was tested with six aircraft (in simulation).

LITA

In August-October 2005, an IDEA-based executive [Baskaran et al. 2006] was deployed on the Zoe rover in the context of the "Limits of Life in the Atacama Desert (LITA)" field campaign, sponsored by the NASA ASTEP program. In LITA field experiments, Zoe autonomously traversed the desert, and using a suite of science instruments, searched for evidence of microscopic life.

Mission goals were specified on a daily basis by scientists working in a remote location (in Pittsburgh, PA), and were then uplinked to the rover operating in the Atacama desert via satellite. Examples of high level goals are motion to a specific location or an area of interest to gather scientific data and deployment of a plow to dig a shallow trench to collect information about sub-surface elements. Given the high level goals, an on-board Mission Planner generated suitable courses of actions for the day that the Rover Executive then executed. In addition to ensuring that the plan generated by the Mission Planner was executed in a timely manner, it was the responsibility of the RE to coordinate recovery sequences whenever faults were triggered. Invariably, the first action the RE had to take in the event of a fault was to quickly stop the rover. The RE successfully handled recoveries for faults such as Navigator, Vehicle Controller and Position Estimator process crashes, and navigation related faults (e.g., inability to find suitably safe drive arcs). The RE also operated in coordination with a Science Planner/Science Observer running on-board to facilitate opportunistic science (science-on-the-fly).

Figure 4: Zoe Rover in the Atacama Desert

Under the control of the Rover Executive, Zoe traversed a cumulative distance of 117 kilometers with a longest uninterrupted autonomous operation of about 5 km. The cumulative time of operation of the rover under the supervision of the RE was 116 hours. The rover autonomously recovered from eight different types of faults during field operations.

Related Work

The Remote Agent (RA) [Muscettola et al. 1998], which operated aboard the Deep Space 1 spacecraft, embodied a three-layer architecture - Functional, Execution and Planning/Scheduling. Each layer called on different reasoning technologies and representations, and was therefore difficult to integrate and test. Most of the traditional autonomous control architectures follow this approach, differing in the degree to which one layer dominates over the others [Locke 1992, Bonasso et al. 1997, Borrelly et al. 1998, Estlin et al. 2000]. They all share the same disadvantages.

More recent approaches, such as CLARAty [Volpe et al. 2001], try to overcome some of the drawbacks of these using an architecture with two layers: a functional layer and a decision layer. The decision layer integrates planning and execution through a tightly-coupled database that synchronizes planning and execution data from two different representations: one from CASPER [Chien et al, 2000] (planning) and the one from TDL [Simmons and Apfelbaum, 1998] (execution). In contrast, IDEA unifies planning and execution under the same modeling and planning framework.

Another two-layer architecture is the model-based approach pursued by Williams [Williams et al 2004]. This framework consists of a Reactive Model-Based Programming Language (RMPL) and an executive (Titan). RMPL is a rich language, given in terms of Markov Processes that combines ideas from reactive programming with probabilistic and constraint-based modeling. The language abstracts the state of the functional level so the programmer can reason in terms of state variables not directly corresponding to observable or controllable states. By comparison, in IDEA, models represent the functional layer variables directly, alongside the representation of internal agent variables and variables of other decision agents. Titan generates control sequences to move the system from the current state to the desired state and diagnoses anomalous situations. It has two main components: a deductive controller which estimates the most likely current state of the system and generates commands to reach the goals, and a control sequencer that produces a set of goals relying on the control program and the current state estimates. In this way the control system design is simplified. Whereas IDEA provides mechanisms for interleaving and controlling deliberative and diagnostic processes, it is not clear in the Titan approach how the underlying cost of diagnosis/planning can be controlled.

Conclusion

Contrary to traditional autonomous control architectures, IDEA unifies the technologies, state representation and domain modeling for deliberation and execution. Under IDEA, all decision makers are planners, operating over different temporal horizons and different breadths of scope. They reason about and act upon a single representation of the state. All relevant features of the application domain, including the controlled subsystem, control strategies and processes, and interactions with other controlling systems, are modeled, intermixed and represented equally in the domain state. This collective state model defines how internal and external stimuli (events, status, data) and the passage of time alters the state, including the initiation, reconfiguration or termination of control actions. This holistic view of autonomous control enables consistency, coordination, flexibility and sophistication that is not possible in traditional three-layer architectures.

IDEA's reliable operation on several space- and military-relevant systems attests to its fundamental approach to autonomous control, and demonstrates how the IDEA architecture can readily be adapted to a wide range of pre-existing system configurations and requirements.

References

[Baskaran et al. 2006] Baskaran, V., Muscetolla, N et al., "Intelligent Rover Execution for Detecting Life in the Atacama Desert", Workshop on Spacecraft Autonomy, AAAI Fall Symposium, Washington D.C., October 2006.

[Bonasso et al., 1997] Bonasso, R. P., Firby, J., Gat, E., Kortenkamp, D., Miller, D., Slack, M. "Experiences with an Architecture for Intelligent, Reactive Agents," Journal of Experimental and Theoretical Artificial Intelligence, 9, 2/3, pp:237—256, 1997.

[Borrelly et al. 1998] Borrelly, J. Coste-Manière, E., Espiau, B., Kapellos, K., Pissard-Gibollet, R., Simon, D. and Turro, N. 1998. "The ORCAD Architecture". International Journal of Robotics Research, 17(4).

[Estlin et al. 2000] Estlin, T., Rabideau, G., Mutz, D., and Chien, S. 2000. "Using Continuous Planning Techniques to Coordinate Multiple Rovers". Elec. Trans. on AI, 4, pp: 45-57.

[Jónsson et al., 2000] Jónsson, A., Morris, P., Muscettola, N., Rajan, K. and Smith, B. 2000. "Planning in Interplanetary Space: Theory and Practice," Proceedings of the 5th International Conference on AI Planning and Scheduling, pp. 177–186.

[Locke 1992] Locke, C. "Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives". The Journal of Real-Time Systems, 4(1):37—53, 1992.

[Muscettola et al. 1998] Muscettola, N., Nayak, P., Pell, B., and Williams, B.C. 1998. "Remote Agent: To Boldly Go Where No AI System Has Gone Before," Artificial Intelligence 103(1-2), pp: 5-48.

[Pedersen et al. 2006] Pedersen, L., Clancey, W., Sierhuis, M., Muscettola, N., Smith, D., Lees, D., Rajan, K., Ramakrishnan, S., Tompkins, P., Vera, A., Dayton, T., "Field Demonstration of Surface Human-Robotic Exploration Activity," AAAI Spring Symposium, 2006.

[Volpe et al. 2001] Volpe, R., Nesnas, I., Estlin, T., Mutz, D., Petras, R., Das, H. "The CLARAty Architecture for Robotic Autonomy," Proceedings of the 2001 IEEE Aerospace Conference, Big Sky, MT, 2001.

[Williams et al 2004] Williams, B., Ingham, M., Chung, S., Elliott, P., and Hofbaur, M. 2004. "Model-based Programming of Fault-Aware Systems." AI Magazine, 24(4), pp. 61-75.