An Architecture for Intelligent Management of Aerial Observation Missions

Michael Freed
NASA Ames Research Center

Pete Bonasso
NASA Johnson Space Center/TRACLabs

K. Michael Dalal, Will Fitzgerald, Chad Frost and Robert Harris
NASA Ames Research Center

Apex, an elaboration of the three-tier type architecture used successfully in many autonomy applications, is designed around the concept of modular reasoning and control services (RCSs) with response-time characteristics as a primary factor in module delineation. We believe this approach reflects a valuable synthesis of requirements from diverse missions types and systems, and avoids pitfalls commonly seen in autonomy architecture design. The paper presents an overview of the architecture, its design rationale and its deployment in Unmanned Aerial Vehicles (UAVs).

Nomenclature

AI	=	Artificial Intelligence
DM	=	Deliberation management
ECI	=	Expected cost of ignorance
M&I	=	Monitoring and interpretation
PDL	=	Procedure Description Language
RCS	=	Reasoning and control service
TC	=	Task control
UAV	=	Unmanned aerial vehicle

I. Reuse of autonomy software

With UAVs rapidly becoming more capable, more available and less expensive, and with good prospects for a regulatory environment favoring more operational flexibility, UAVs are proposed for use in increasing variety of missions. In many cases, autonomy or reduced human supervision is considered a functional or economic requirement[1]. This presents a significant engineering challenge. In particular, as UAV technologies and missions evolve, so must the intelligent automation (autonomy software) used to fly the vehicle, control the payload and manage the mission. Autonomy software, like sophisticated automation software of any kind, is expensive to create and risky to use without extensive testing. One way to mitigate these costs and risks is to reuse software proven in previous autonomy applications.

Reusable autonomy software should have three key properties. First, it should implement reasoning and control capabilities that satisfy important requirements for new applications but would be difficult and time-consuming to reconstruct – i.e. it must supply significant **leverage** in constructing new autonomous systems. Second, the implementation of these capabilities should be easily **portable** to new applications. Third, the set of capabilities should be easily **extensible** – i.e. there should be no significant problems of integration when adding new capabilities or modifying preexisting ones.

The first property, leverage, requires implementing autonomy-related capabilities in a more general way than is strictly required for a particular application. For instance, rather than design software to diagnose anomalous values from a specific vehicle sensor, one could incorporate diagnostic reasoning software[2,3] that applies to a wide variety of sensors and other system components. There are costs to creating and

validating software more general than it needs to be, so the challenge is to spend effort generalizing a solution where the future payoff is greatest. We use the term "reasoning and control services" (RCS) to refer to implementations of specific autonomy-enabling capabilities expected to be useful in a wide variety of applications.

Portability and extensibility depend on the overall software architecture in which RCSs are implemented. A software architecture specifies unifying design decisions about, e.g., primary data types, modules and integration principles[4]. Ideally, these decisions are sufficiently comprehensive to ensure desirable system qualities. Software architectures designed specifically for autonomy (autonomy architectures) have been developed in a variety of application areas including UAVs[5], advanced life support[6], ground mobile robots[7-9] and unmanned spacecraft[10-12] and many others. Each of these systems incorporates reasoning and control capabilities of demonstrated value. However, few such systems have been reused in a variety of distinct applications and little investigation has been made of the specific problems of autonomy software reuse.

We have developed reusable autonomy software with an overall objective of reducing the time, expertise and inventiveness required to build new systems. The software, Apex, has been used in diverse applications including two UAV efforts described below. These efforts have been an invaluable source of lessons learned about the specific requirements for effective reusability, particularly in defining the kinds of reasoning and controls services most often found to be useful in new applications and the specific architectural commitments needed to facilitate reuse. In this paper, we describe Apex RCSs and architecture and illustrate our approach to reuse in the context of two UAV applications.

II. Example applications

A. The Autonomous Rotorcraft Project

Figure 1: ARP RMAX research aircraft (left) and instrumentation trailer (right)

The Autonomous Rotorcraft Project[13] is an Army/NASA collaborative effort to develop UAV autonomy with a subscale rotary wing aircraft used as a demonstration platform. Autonomy-enabling capabilities developed by the project include obstacle avoidance path planning, safe landing area determination in GPS-denied conditions, robust flight control software for takeoff, landing and forward flight and intelligent mission management for diverse missions. The Yamaha RMAX helicopter used by the project is capable of approximately one hour of hover flight duration with a 65lb payload and a maximum speed of about 40kts. Originally designed for remote control agricultural seeding and spraying, the RMAX has been modified for autonomous flight. Modifications include an avionics payload carrying a navigation and flight control computer, a research computer, IMU, GPS receiver and radio communications equipment. Separate from the avionics payload is a vibration-isolated stub wing mounting various tilt-actuated cameras for stereo passive ranging, monocular tracking and storing or streaming color video of an observation target. A SICK PLS scanning laser mounted beneath the nose is used for obstacle avoidance and high-resolution mapping. Hardware and software systems are continuously enhanced with flight tests taking place approximately weekly.

Apex provides mission-level autonomy capabilities supporting a range of aerial observation mission-types, focusing particularly on surveillance missions in which numerous observation targets must be monitored by a single UAV. The objective is to maximize the value (quantity, quality and timeliness) of information about the targets returned to ground systems. For example, in a fire detection application, there may be many structures that could potentially catch on fire but only a single UAV to move from site to site to check for an outbreak. Sites may vary greatly in importance, probability of catching on fire, remoteness from other sites, remoteness from firefighting resources and many other factors that affect the value of observing the target at any given time. The optimum behavior might involve patrolling only a subset of the sites, or visiting some far less frequently than others. Analogous applications in, e.g., disaster management, force protection, Earth science and security present the same fundamental problem. We have formally characterized this general class of missions in order to define specific autonomy requirements and performance evaluation criteria[14]:

$$ECI = \sum_{T}^{\text{Targets}} \sum_{\text{i}}^{\text{Intervals}} \int_{\text{t=t1}}^{\text{t2}} p(t) \cdot Cost(t2 - t)dt$$

We focus on surveillance missions in which events of interest are rare. Mission performance cannot be evaluated effectively on the basis of events actually observed, but must instead be characterized in terms of how well the agent reasoned about the probability and costliness of events that might have occurred. This entails an essentially decision-theoretic approach. Given that the vehicle can generally observe only a single target at a time and must spend time transiting to and examining each target, it will necessarily not be observing most targets most of the time – i.e. it will be “ignorant” of the state of these targets. We define Expected Cost of Ignorance (ECI) for the period between successive observations of a given target as the sum, for all time points t in the interval, of the probability of an event occurring at t multiplied by the cost if it occurs at t. Summing for all such intervals and for all targets, we can compute a total ECI for the mission given a specific target observation schedule. The goal of an autonomous agent is to minimize overall ECI accumulated over the time period of a mission.

Autonomy requirements for such missions derive from three sources: the objective function (metric) above, characteristics of targets and environment needed to define specific probability and cost functions (e.g. target value as a parameter of the cost function) and characteristics of vehicle, sensors, physical environment and ground systems that constitute sources of uncertainty or constraint in executing observation plans.

Apex incorporates a range of reasoning and control capabilities aimed at meeting these requirements. A periodic-surveillance planning service generates plans specifying visit order and observation actions at targets. In fact, as discussed further on, there are several such planning services which vary in situational effectiveness. A solver selection service is used to predict which planner will perform best in a given situation. Services supporting contingency detection can determine, e.g., when weather or other conditions have changed enough to warrant replanning or when operational requirements have been violated and require corrective action. Contingency detection behavior can be specified directly by a developer or derived from a TEAMS model[15].

Apex computes at-target observation behavior based on specific data-acquisition goals. For example, it can command the vehicle to arc over a target, controlling camera tilt to maintain target track through the maneuver. As the vehicle nears the peak of the arch, beyond which the camera can no longer be tilted enough to stay pointed at the target, the vehicle slows, changes its heading 180 degrees and then increases speed, completing the arch in reverse in order to maximize tracking. In general, at-target behaviors involve coordinating three-dimensional flight-path (e.g. straight over, straight aside, spiral, arc, sweep), attitude mode (coordinated flight, fixed attitude, pointing), speed profile, sensor payload behavior and data handling behavior. Apex synthesizes these behaviors at runtime rather than requiring an operator to specify specific behavior directly.

B. The Intelligent Mission Management Project

Figure 2: Predator B aircraft (left) and illustration of use in Western States Fire Mission (right)

The Intelligent Mission Management project supports autonomous UAV-based acquisition of Earth science data. Platform requirements are expected to vary from small fixed-wing aircraft to large, high-altitude, long endurance aircraft able to carry large, sophisticated sensor payloads. At present, an initial set of autonomy capabilities have been developed, integrated with collaboration-facilitating ground systems[16] and demonstrated in two simulation environments. Initial flight testing is scheduled to begin in 2007 on a Predator B, a fixed-wing aircraft with approximately 3000 lbs payload capacity and range exceeding 3000 nm.

In the nearer term, autonomy capabilities have been developed for the Western States Fire Mission, a demonstration in which airborne fire sensing payload will be used to map a large number of fires (possibly several dozen) over a period of approximately 24 hours (see Refs. 17 and 18 for a more complete description). The actual flight test will be conducted with human remote pilots rather than autonomous control. Apex will fly a simulated version of the mission in parallel as a demonstration and may provide decision support to human operators for flight plan generation. The autonomy performance metric for this mission, and for anticipated IMM missions in general, has been formally characterized as follows:

$$\overbrace{M}^{MissionScore} = \overbrace{\left(\sum_{j=1}^{n} v_j c_j r_j s_j o(t)_j\right)}^{Benefits} - \overbrace{\left(\sum_{k=1}^{m} (H_k \cdot W_k) + P \cdot (R \cdot C_v + M \cdot C_m)\right)}^{Costs}$$

The primary goal of IMM missions is to acquire as much high quality data as possible within a timeframe defined by vehicle endurance or a specified maximum duration. Each observation target is associated with a value (v) and a set of data quality preferences for image resolution (r), clarity (s), coverage (c) and timing (o). For example, it may be desirable for a particular target to get 10m per pixel image data (requiring a certain maximum altitude), prevent image gaps (requiring wings-level attitude while observing), acquire data on the perimeter but not necessarily the interior of the target area, and to synchronize observation of the target with a MODIS satellite overpass. If these preferences are not fully met, the value of having observed the target is discounted from v.

The autonomy software is responsible for maximizing information benefits by generating and executing an optimal flight plan. Implicit in the metric are tradeoffs that the autonomy may need to reason about – e.g. which subset of the targets to visit, whether to sacrifice resolution to gain clarity by ascending out of a turbulent altitude and whether to visit one less target in order to recover from wings-level violations at the current target by reflying certain segments. In addition, plan validity may be constrained by several factors including target visit ordering constraints, airspace restrictions and a requirement to remain within some maximum distance of an emergency landing locale.

A second role of the autonomy is to minimize costs. *Personnel costs* depend on the number of people with an operational role in the mission and the time and pay rate of each. This accounts for a basic economic contribution of increasing autonomy: reducing the need for human operators, particularly those who cost the most or have the rarest and most in-demand skills. *Failure costs* result from unanticipated losses such as damage to a sensitive optical sensor accidentally pointed at the sun, or aircraft loss resulting from a critical system failure. Autonomy software can minimize failure costs both by avoiding bad decisions that cause failure and by responding quickly and correctly to failures in other systems.

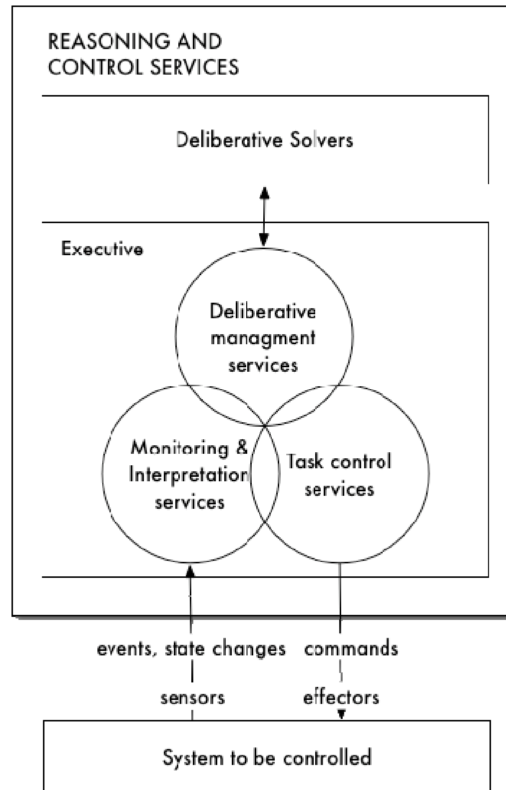

Figure 3. Overview of Apex integration architecture.

IMM missions and ARP missions have in common that they involve autonomous aerial observation of separate targets and that the information value from a particular mission can be formally characterized. Many of the specific reasoning and control services provided by Apex are used in both projects including services supporting contingency detection, generation of at-target observation maneuvers and planner invocation during the mission (replanning). However, the overall mission objectives are quite different. In ARP missions, the core planning problem is to reason correctly about observations whose value is defined probabilistically and with respect to the time since the target was last visited. IMM missions are focused on sequencing single visits to selected targets with known (assigned) value and on maximizing data quality at a given target. In addition, plans for IMM missions may require meeting constraints such as ordering constraints on target visits or synchronizing an observation with a MODIS satellite overpass. Mission planning requirements in the two applications are thus quite different.

III. Building Apex applications

Constructing a new autonomy application using Apex involves two main steps. The first is to integrate with the controlled system (see Figure 3). Integration requires specifying system properties (e.g. real-time vs. simulated-time) and then implementing data transport mechanisms allowing the Apex-defined subsystem to receive information from and send commands to other subsystems. Lower layers of the transport mechanism (physical and media layers and session layer in the OSI model) are typically managed using vendor-supplied interfaces, such as TCP or UDP over IP. In many cases, software filters and transducers must be implemented to manage data flow into Apex and to convert the input data into a form Apex can process. For example, the ARP avionics software publishes a message packet, RCDynamicState, containing information on more than 30 state variables including, e.g., airspeed, heading, altitude and roll-rate. Our integration code subscribes to RCDynamicState, extracts the values relevant to the autonomy software, and then converts into an Apex-usable format.

Commands output from an Apex agent to the controlled system are referred to as *primitive* actions. Such actions are generated using a template. For example, the primitive template below defines how the system can command a flight simulator (XPlane) to begin broadcasting information on a specified aircraft

state variable via UDP. The `index` clause defines the name of the command and its parameters. When the form is invoked, code embedded in the `on-enablement` clause is executed to enact the command.

```
(primitive
 (index (enable measurement ?variable))
 (on-enablement
  (xplane-set-channel
   (xplane-channel-lookup ?variable) "DSEL")))
```

The `primitive` form is one of several top-level forms in Apex's Procedure Description Language (PDL). Apex reasoning and control service modules define the operational semantics of PDL constructs. For example, the `primitive` form above illustrates two basic Apex RCSs: *pattern matching*, and *task-state transition control*. The pattern matching service links an Apex agent goal to a construct (such as a primitive form) specifying how to achieve that goal. This allows for a flexible naming scheme for PDL forms that enhances readability and can be parsed by RCSs into structured goal representations. In addition, the pattern matcher can enforce a wide range of match constraints. For example, by replacing the index clause in the previous example with the one below, the primitive form would automatically check whether the requested XPlane variable is legitimate.

```
(index (enable measurement (?is ?variable valid-variable-p)))
```

The second step in constructing an Apex application is to specify desired autonomous behavior. Apex agents are goal-driven – i.e. they are given a set of goals to be accomplished rather than specific action sequences to carry out. Reasoning and control services determine how to achieve these goals based on general knowledge represented in PDL *procedures*, specific situation knowledge received from sensors and communication channels and a set of *tasks*. A task represents an intention for how and when to pursue a particular goal. More specifically, it represents a *commitment to act under specified conditions to realize a policy.* Such conditions may represent normal occurrences or states (e.g. current altitude above ground level is high enough to safely perform a maneuver) or unusual, off-nominal conditions (e.g. altitude has been in uncommanded decline for 20 seconds). Apex handles nominal and off-nominal conditions in the same way. This simplifies behavior specification and allows contingent behaviors to take advantage of the full range of task RCSs. A *policy* defines the specific condition that the task is intended to bring about or preserve. In some cases, the condition is a goal as the term is often used in the artificial intelligence literature; *i.e.*, a state to be achieved, maintained or prevented, *e.g.* "maintain altitude FL120". The desired condition might also be more abstract. For instance, it may be tied to indefinite (non-specified) state variables (e.g. complete checkout procedure); it may be defined functionally (*e.g.* be vertically separate from any aircraft with 5 miles by at least 1000 feet); or it may be defined by an optimization criterion (*e.g.* be at the best observation position for current target given sun-angle, wind-vector and optimization function *F*). We use the term policy to include all such cases.

Information about how to accomplish a task – i.e. how to realize a task's policy – is represented in PDL procedures. The following example defines a procedure to image a ground target using a high-resolution fixed-angle camera called `camera-1`.

```
(procedure
  (index (get hires image ?target))
  (profile camera-1)
  (step s1 (move-to best-imaging-loc for  ?target => ?loc))
  (step s2 (power-up camera-1))
  (step s3 (orient-camera-to ?target)
      (waitfor (:and (ready camera-1)(location +self+ = ?loc))))
  (step s4 (take-picture camera-1)
     (waitfor (end ?s3)))
  (restart-when (task-state +this-task+ = resumed))
  (end-when (image-in-memory ?target))
```

Each procedure contains at least an `index` clause and one or more `step` clauses. As with primitives, the `index` uniquely identifies the procedure and defines what kinds of goals it serves. The pattern matching service is used in procedure selection, just as it is for primitives. Each procedure step specifies a new task (subtask) to be instantiated. These subtasks are not necessarily carried out in listed order. Instead, they are assumed to be concurrently executable unless otherwise specified. If ordering is desired, a `waitfor` clause is used to specify that the completion of one step is a precondition for the start of another. For instance, the steps labeled `s1` and `s2` do not contain `waitfor` clauses and thus have no preconditions; tasks generated from these steps can begin execution as soon as the procedure is invoked and can run concurrently. Step `s3`, in contrast, includes the clause `(waitfor (:and (ready camera-1) (at-location ?loc))`. This means that a `s3` becomes eligible for execution (enabled) only when the agent received messages indicating a successful outcome to `s1` and `s2`.

Tasks can be in any of the following states: pending, enabled, ongoing, interrupted, terminated. A task with preconditions specified by a `waitfor` clause starts in a *pending* state. When the preconditions are satisfied (or initially if there are no preconditions) the task becomes *enabled*. Tasks often require resources (declared in a `profile` clause as above) and may conflict with other running or enabled tasks with overlapping resource requirements. A task must wait in an enabled state until it becomes the highest priority competitor for all needed resource; at that time its state is changed to *ongoing*. In some cases, an ongoing task becomes lower priority than a competing task in which case its state becomes *interrupted*. Finally, when the task completes (or is aborted), its state is changed to *terminated*. Numerous Apex reasoning and control services are involved in managing task state transitions. For example, there are services for *detecting conflicts*, *resolving conflicts* and *handling any interruptions or resumptions* that result. These are discussed in detail elsewhere[19-21].

Other important services check whether conditions requiring a task state transition have been met. Conditions, defined in `waitfor` clauses and other PDL constructs, are represented in a task subcomponent called a *monitor*. Monitors check for their associated conditions using three sources of information. First, internal mechanisms (RCSs) generate events. For example, an event is generated when a task terminates, thus satisfying a precondition for any task with an associated ordering constraint. Similarly, any resources allocated to a terminated task are released, causing any task waiting for that resource to compete for it. Second, external sources provide information via sensors and communication channels. Third, information from these other sources is stored in state variable histories. These are essentially time-stamped measurements for a specific attribute of a specific object (e.g. the altitude of a particular aircraft). *Condition monitoring* services check these sources to determine whether information from some or all of these sources can be interpreted as satisfying a specified condition. Conditions may be defined in terms of constraints on individual state variable values, episodes defined abstractly over a state variable interval, logical constraints, and Allen predicates on intervals.[22] More detailed information on condition handling is included in Appendix B.

When task preconditions are satisfied and the task becomes enabled, *selection services* determine how the task policy will be carried out. The selected method is later (when the task becomes ongoing) used to create new specific new structures to enact the policy; these structures are referred to as a *controller*. This process can occur in any of several ways. One possibility is that a procedure will be selected, leading eventually to further (hierarchical) subtask creation. For instance, a task T1 with the goal `(power-up camera-1)`, created using the example procedure above, might be matched against another procedure with `(power-up ?any-camera)` as its index clause. That procedure would then provide a template for creating new tasks that, in combination, accomplish the power-up task. A second possibility is that the task will be matched to a primitive, which defines a primitive controller for the controlled system (possibly just a single command as in the earlier example).

If no matching procedure or primitive exists, the selection RCS will call an AI planner (solver) to construct a new procedure. Planners are valuable for handling goals not handled by anything in an agent's procedure library. For some kinds of goals, use of a planner is the norm (e.g. path planning in non-routine task environments). However, procedures can be used to define a wider range of behaviors than is practical with current-generation planners and they more easily facilitate meeting agent response-time requirements. Predefined procedures are therefore used where possible.

Specifying the procedures, primitives and planners that define autonomous agent behavior is typically the most time-consuming and effortful part of applying Apex in a new application. In practice, making Apex reusable requires limiting these costs. One approach is to make the behavior specification language

as usable and convenient as possible. In particular, the language should be readable (e.g. compact, not excessively abstract), expressive (the right way to express behavior information should be obvious), intuitive (the actual behavior should conform to the intended behavior) and should have clear correspondence with generated runtime structures that may need to be examined when debugging. We have tried to develop Apex's Procedure Definition Language to intuitively mirror the reasoning and control services provided. User feedback has been invaluable in learning how to improve PDL usability, particularly in discovering non-intuitive language constructs and failures to provide sufficiently expressive notation. Second, powerful visualization tools can be created to help understand the online decision-making of an autonomous agent and to facilitate debugging. Apex's Sherpa development tools serve this role[24]. Finally, it is sometimes possible to reuse behavior specifications such as PDL procedures in multiple applications. This is especially valuable if specialized expertise was required to construct the representations – see Ref. 25 for examples.

IV. Apex Reasoning and Control Services

In Apex, Reasoning and Control Services are, specifically, the set of functions that create tasks, modify tasks or provide procedural semantics for task monitors and controllers. Several capabilities implemented as distinct RCSs have been mentioned already including, e.g., matching sensor inputs to precondition patterns and detection of resource conflicts. The functionality and degree of integration of system RCSs are largely hidden from a developer using Apex, but these attributes are central in determining the system's value and fitness for reuse.

RCS functions in Apex fall into four general classes. **Monitoring and interpretation** (M&I) services provide functionality for handling input data (e.g. from sensors) and stored data. This includes creating monitors from PDL specifications, pattern matching to asynchronous messages containing data, logging and retrieving data, enforcing policies logged data quantity, extracting, subsampling, interpolating, trend-finding, and many others. In general, M&I services are used to create and perform analysis prescribed by task monitors. **Task control** (TC) services provide functionality for deciding and enacting behavior. This includes, e.g., selecting procedures and primitives, instantiation of new controllers, detecting resource conflicts, resource allocation between contending tasks and governing task state transitions (enablement, disenablement, interruption, etc…). TC services are for creating and carrying out actions prescribed by controllers.

As discussed in the next section, TC and M&I services are integral parts of the central (executive) decision-making component which is designed to be both general-purpose and rapidly responsive. **Solvers** are services that perform either M&I or TC functions, but use algorithms (e.g. AI planners) that are useful, perhaps critical for a given application, but either too specialized or too computationally expensive for inclusion in the executive. **Deliberation management** (DM) services incorporated into the executive provide functionality related to the use of solvers: selection, invocation, process control and transformation of output into a form usable by executive functions.

Within these general classes, Apex RCSs can be further divided into a hierarchy of more specific types. The leverage Apex provides for constructing new autonomy applications depends on the functionality and integration among specific RCSs. Our approach to maximizing leverage is illustrated by two closely related examples.

Example 1. In the Intelligent Mission Management application described earlier, the sensor payload requires wings-level flight over a target. When violations of sufficient magnitude and duration occur, the sensor cannot obtain data on a portion of the target area. The PDL example below defines a procedure step that responds to a wings-level violation by creating a task to repeat the flight segment where the loss occurred.

The `waitfor` clause in this example leads to creation of a monitor that looks for a wings-level violation likely to produce data loss. In particular, it looks for an episode (a durative condition) in which the roll-angle of the aircraft exceeded the threshold `+max-roll-angle-deviation+` for a period of time greater than `+max-off-nominal-roll-angle-duration+`. Both time and angle thresholds are established at runtime using functional variables (indicated by the surrounding + symbols). When a violation is detected, the `select` clause determines the location where the violation episode began, then passes this value to the step activity description where it is used to specify a task to refly the interval.

```
(step (refly segment visited from ?loc to +current-location+)
      (waitfor
       (:episode roll-violation (roll-angle +this-aircraft+)
         :value (> +max-roll-angle-deviation+)
         :timing
          (:duration (> +max-off-nominal-roll-angle-duration+)))
         :quality (:no-constraints)))
      (select ?loc
        (most-recent-SVvalue 'location +this-aircraft+
          :prior-to (start ?roll-violation)))
```

Example 2. In the Autonomous Rotorcraft Project, there is a requirement for the vehicle to stay in communication with a ground station. Signal strength from ground normally varies and can even drop to zero for short periods without constituting a violation of the communication requirement. When a violation does a occur, the vehicle should return to a position where signal strength was acceptable, signal to ground that a violation occurred and await instructions.

```
(step (regain-comm-at-location ?location)
    (waitfor
        (:episode comm-violation (signal-level 900MHzModem)
          :value (< (+ +nominal-signal-strength+))
           :timing
            (:duration (> +max-off-nominal-signal-strength-duration+))
           :quality (:no-constraints))
    (select ?location
     (most-recent-SVvalue 'location +this-aircraft+
      :prior-to (start ?comm-violation))))
```

The `waitfor` clause in this example defines a monitor that looks for a communication dropoff lasting more than `+max-off-nominal-signal-strength-duration+`. When this occurs, the last location where signal strength was acceptable is determined and the value used to specify a task returning to that position.

The two examples illustrate the reusability of several reasoning and control services in different applications. In both cases, the agent is looking for an interval in which some measured quantity goes out of bounds for an interval. This involves M&I services for monitoring, logging and retrieving sensor data and for rapidly detecting the establishment of conditions (episodes) defined by temporal properties and value trends. In both cases, a property of the episode is used to compute part of the response. This involves TC services for finding or interpolating within data with specified temporal and relational properties and for handling information dependencies in specifying enabled tasks.

None of these services were part of the original version of Apex. Initially, the system could monitor only for individual events matching specified patterns and for conjunctions and disjunctions of such events. Over a series of application efforts, functional limitations of M&I services often proved problematic, resulting in a lengthy list of desired enhancements.[26] In some cases, the need for enhancement arose from integration issues – i.e. the need to make new functionality work correctly and flexibly (composably) with other RCSs. In other cases, implementing the functionality required specialized expertise or substantial engineering effort to achieve acceptable performance. In general, it has proven hard to guess which functional capabilities are most reusable and thus most worth investing effort in. Lessons learned from building applications have proven the only reliable guide.

V. Apex Software Architecture

Capabilities of implemented reasoning and control services define potential *benefits* from reusing autonomy software. Limits on portability and extensibility determine how difficult the software will be to reuse, and thus determine *costs*. A well-designed software architecture, by imposing unifying design

decisions on data representation, modularity and integration, can go a long way towards facilitating portability and extensibility.

The Apex architecture is based on a well-known approach to autonomy architectures in which functionality is divided into three modular **layers** (See Refs 27,10 and compare to Ref 7). Typically, the top layer contains a computationally expensive AI **planning algorithm** that sends plans to a fast-acting reactive **execution system** (layer 2). The execution system dispatches runtime commands to platform- and domain-specific **skills** (layer 3), which directly control subsystems. Mission-level goals, constraints and preferences are fed to the planner from an external source, either a human operator or a software **mission manager** component. In Apex, the mission management and execution system functions are combined into a single component referred to as the **executive**. By analogy to piloted aircraft, the skills layer can be thought of as implementing traditional flight automation (autopilot) while the planner and executive play the role of a human pilot.

The three-layer approach makes two design concepts paramount. The first is to separate "deliberative" functions that may be too computationally expensive to meet application response-time requirements (layer 1) from more responsive functions (layers 2 and 3). The central importance of response time for UAV applications is easily understood. For example, the mission-level objective for both UAV applications described earlier involves optimizing the overall value of information returned to ground personnel. Plans that are (approximately) optimal when generated can become invalidated or less optimal by a wide variety of execution-time occurrences – e.g.: changes in wind, unexpectedly long or short time required to carry out a plan step, additions, deletions and changes in priority to the target set, and new information affecting the likelihood or expected importance of changes at particular targets. Frequent replanning should be expected.

However, algorithms able to generate optimal plans (solver RCSs) are typically very demanding of computational resources – NP-complete or worse. There is no practical upper limit to the amount of time such algorithms might take before finding the best plan or even an acceptable plan. Separating out expensive deliberative functions, allows functions in the 2nd and 3rd layers to operate as a responsive outer-loop control system for the UAV. When deliberative processes take too long, default or safing behaviors can be invoked. There are several ways to make use of deliberative planning processes given a responsive execution system. The most common approach is to generate plans only prior to a mission, or else at scheduled time-points in the mission where responsiveness requirements are expected to be low (See Refs. 6 and 28). Alternately, deliberation can be treated as a controlled process that can be paused, redirected and, in the case of anytime algorithms (see, for example, Ref 28), polled for a "best result so far" when a solution is needed.

The second design concept intrinsic to the three-layer approach is to separate the most-reusable code (the top two layers) from the least reusable (the bottom layer), thereby facilitating portability. Determining which code is reusable is partly a matter of identifying application-specific problem features and either decoupling code that assumes these features from code than does not, or else designing code that generalizes over such features and can be configured for their presence or absence. The importance of identifying and separating out problem-specific features is illustrated by the Ariane 5 incident[29] in which an assumption built into reused code led to mission failure. The functional scope of Apex RCSs represent decisions about how to distinguish general from application-specific capabilities. Portability is further enhanced by well-designed, clearly documented APIs. The Apex API consists mainly of a configuration function (`defapplication`), two functions for providing input to an Apex agent (one for message-passing, one for publish-subscribe) and the PDL primitive form for specifying Apex agent output.

Achieving good extensibility requires addressing issues beyond those specifically considered in the three-layer approach. First, extension is simplified if RCSs are built on a common foundation, particularly if their inputs, outputs and internal states are represented using a small, shared set of well-defined structured information types. In Apex, all state information able to influence the behavior of an Apex agent is held in tasks (including monitors and controllers) and a few less central structures (histories, resource allocation tables, agents).

Second, it is generally easier to extend a system organized around a fine-grained, modular decomposition of functions than one whose modules incorporate numerous, closely coupled functions. A smaller functional grain size makes it more likely that swapping in a module with extended capabilities will not be disruptive and more likely that an entirely new functional capability can be implemented in a module and integrated the same way as some similar, pre-existing module. The hierarchical structure of Apex RCSs is designed to provide fine-grained modularity. For example, M&I services for detecting cyclic

patterns are currently being designed; integration should require little more than a straightforward insertion of the new RCSs into the executive and a simple extension to PDL's syntax definition.

Third, if the need for particular kinds of extensions can be anticipated, the architecture can be designed to facilitate incorporation. For example, current Apex M&I services can detect diverse event-pattern types including certain trends and combinations of simpler patterns with specified logical and temporal relations. Requirements to further extend the kinds of recognizable patterns (e.g. cyclic patterns) are very likely. The modularity and integration approach within M&I software has been designed with this expectation, particularly in associating different classes of detection with distinct modules, providing generic operators for handling data (state variable) histories and streamlining the process for extending the PDL language.

C. Extensibility example

Anticipating needed classes of extension is especially important for applications where requirements are either not well-understood at the outset or are likely to evolve during the operational lifetime of the autonomous system. In the Autonomous Rotorcraft Project, for example, the objective has been to provide a very flexible capability for using a single UAV to maintain situation awareness at spatially separated sites. Both NASA and the U.S. Army, joint sponsors of the project, carry out missions and routine operational activities that fit this general description. However, there is no well-defined concept of operations for using autonomous UAVs in this capacity.

To provide a capability flexible enough for missions that are not currently specified and likely to evolve, we have formally characterized the *surveillance planning problem* as previously described (see also Ref 30). A surveillance mission, in this context, is defined by a set of observation targets and functions for each target for determining the time-varying likelihood that an important event will occur, the time-varying cost of not observing an event once it has occurred, and the time-cost of observing a given target to detect the event of interest. The role of a surveillance planner, implemented as a solver RCS since this capability is not provided natively by Apex, is to find a plan that minimizes the sum ECI (expected cost of ignorance) for all observation targets.

Surveillance missions of this sort will vary in different ways. Some will have a greater number of targets and some fewer. In some, the targets will be close enough together that a detailed aircraft model will be required to compute transit times while in others, point-to-point distance provides an accurate estimate. Targets may fall into spatial patterns that simplify the planning problem (e.g. clusters) or may be uniformly distributed. Targets may be of equal value or value might vary widely. Constructing a single planner that produces good surveillance plans in all conditions presents a difficult, perhaps insurmountable challenge.

An alternative is to develop a variety of surveillance planners with varying strengths and weaknesses and RCSs able to select the best planner for the current mission. There are four primary elements to this approach: invocation, problem classification, planner selection, and translation. Invocation of a planner occurs when the monitor component of a task determines that all preconditions have been met – i.e. that the task is eligible to start. The selection RCS then tries to find a stored procedure whose index clause matches the task's policy specification. If one is found, it is then used to generate a controller to carry out the policy. Otherwise, the solver selection RCS looks for a planner able to generate a new procedure.
Sometimes there are multiple planners that can do the job. In such cases, problem classifier RCSs are used to compute features of the planning problem useful for deciding between alternative planners. In the case of ARP surveillance, we have implemented two planners that take very different approaches. The first planner looks for the best repeatable cycle using a modified 2-Opt Traveling Salesman Problem algorithm to quickly select a path for a given (sub)set of targets. The second planner (WAM) uses best-first search with replacement, using a heuristic based on target proximity and obsolescence (time since last visit) to incrementally construct a full mission plan.

To determine which plan to use, the space of possible surveillance missions is defined in terms of five features: number of targets, spatial distribution of targets, variability of target importance, variability of target cost accumulation rate and mission area size. Given a current mission (planning problem) described in terms of these features, the solver selection RCS consults a predefined performance profile table (PPT) to select the best planner. The example PPT (figure 4) shows how the two algorithms compare in a wide

Min of Best			Count	Space							
			4			8			16		
Scale	Rate	Cost	2-Cluster	Globular	Uniform	2-Cluster	Globular	Uniform	2-Cluster	Globular	Uniform
Large	Clustered	Clustered	1	1	0	3	3	1	1	3	1
		Fixed	1	1	0	3	3	1	2	3	2
		Uniform	1	2	1	1	2	1	1	1	1
	Fixed	Clustered	1	1	0	1	1	1	1	3	1
		Fixed	1	1	0	3	1	1	2	3	2
		Uniform	1	1	1	1	2	1	1	3	1
	Uniform	Clustered	1	1	2	1	3	1	1	3	1
		Fixed	1	1	2	3	3	1	2	3	1
		Uniform	1	1	1	1	2	1	1	3	1
Medium	Clustered	Clustered	2	2	0	1	2	2	2	2	2
		Fixed	1	2	0	2	2	0	2	2	2
		Uniform	1	2	2	2	2	2	2	2	2
	Fixed	Clustered	1	2	0	2	2	0	2	2	2
		Fixed	1	2	0	2	2	0	2	2	2
		Uniform	1	2	2	2	2	2	2	2	2
	Uniform	Clustered	1	2	1	2	2	2	2	2	2
		Fixed	1	2	2	2	2	2	2	2	2
		Uniform	1	2	0	2	2	2	2	2	2
Small	Clustered	Clustered	2	1	0	2	0	0	2	2	2
		Fixed	2	1	0	2	1	0	2	2	2
		Uniform	2	2	3	2	2	2	2	2	2
	Fixed	Clustered	2	1	0	1	2	2	2	2	2
		Fixed	2	1	0	2	2	2	2	2	2
		Uniform	2	1	0	2	1	0	2	2	2
	Uniform	Clustered	2	2	0	1	2	2	2	2	2
		Fixed	2	2	0	1	2	2	2	2	2
		Uniform	1	1	0	2	2	2	2	2	2

Human
2-Opt
WAM
no diff

Figure 4. Surveillance planner performance profile

range of conditions.[1] Once a planner is selected and run, the resulting plan can be transformed into a PDL procedure. Apex treats stored and newly generated procedures uniformly, using the new procedure to specify behavior as if it had been present all along in the stored library.

The described RCS components for invocation, classification, selection and translation define an extensibility framework. When a new, distinctively capable planner is created, integration only requires creating an invocation interface (equivalent to a PDL index clause) for the planner, running it against whatever test suite was used to create the PPT, then updating the PPT to indicate when the planner outperforms alternative methods. Similarly, if additional selection features are desired (e.g. to select a planner based on how well it performs after a given maximum deliberation time), a developer needs to create a classifier for the feature, modify the selection RCS to call the classifier and then update the PPT with appropriate feature values. Creating a new classifier can be simple or very difficult depending on the feature and some analysis may be required to determine a useful range of feature values; the rest of the process can be automated.

VI. Conclusion

Reuse of autonomy software is particularly important for UAVs. Given the increasingly central role of autonomous operation in new UAV mission concepts and the resulting need to implement such software for rapidly evolving platforms and payloads, software reuse could be enormously helpful both for managing development costs and for preventing software implementation and validation from becoming a primary bottleneck in development of new systems. However, reusing autonomy software requires meeting specific, practical challenges that have not previously been articulated and examined in detail. Our focus in this paper has been to identify critical challenges and to highlight elements of our approach, implemented in Apex, that have proven valuable in UAV projects and other applications.

[1] Note that this version of the table also shows where human planners perform better than either algorithm, thus providing a principled way to decide if the autonomous system could benefit by requesting human assistance

References

[1] Huang, H.M., Messina, E. and Albus, J. "Autonomy Level Specification for Intelligent Autonomous Vehicles: Interim Progress Report," Proceedings of the 2003 Performance Metrics for Intelligent Systems (PerMIS) Workshop, Gaithersburg, MD, August 16-18, 2003.

[2] Davis, R., Diagnostic Reasoning Based on Structure and Behavior, *Artificial Intelligence*, 24:347-410, 1984.

[3] Sticklen, J., Chandrasekaran, B., and Bond, W. "Distributed Causal Reasoning." *Knowledge Acquisition*, 1:139-162, 1989.

[4] IEEE Std 1471-2000: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Computer Society, 2000.

[5] Ricard, M. and Kolitz, S. "The ADEPT Framework for Intelligent Autonomy." *Proceedings of the Intelligent Systems for Aeronautics Workshop*. Brussels, Belgium, 05/13/2002 to 05/17/2002.

[6] Bonasso, P. Kortencamp, D. and Thronesbery, C. "Intelligent Control of a Water-Recovery System: Three Years in the Trenches," *AI Magazine,* Vol. 24, No. 1 pp. 19-44, March 2004.

[7] Nesnas I.A.,Wright A, Bajracharya M., Simmons R., Estlin T, Kim, W.S., "CLARAty: An Architecture for Reusable Robotic Software," *Proceedings of the SPIE Aerosense Conference*, Orlando, Florida, April 2003.

[8] Muscettola, N.; Dorais, G. A.; Fry, C.; Levinson, R.; and Plaunt, C. "IDEA: Planning at the Core of Autonomous Reactive Agents. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, 2002.

[9] Myers, K.L, *User Guide for the Procedural Reasoning System*, Artificial Intelligence Center, Technical Report, SRI International, Menlo Park, CA, 1997.

[10] Bernard D., Dorais G., Fry C., Gamble E., Kanefsky B., Kurien, Millar J.W., Muscettola N., Nayak P., Pell B., Rajan K., Rouquette N., Smith B., and Williams B., "Design of the Remote Agent Experiment for Spacecraft Autonomomy ", Proceedings of the IEEE Aerospace Conference. Aspen, CO. 1998.

[11] Barrett A., Knight R., Morris R., and Rasmussen R. "Mission Planning and Execution Within the Mission Data System*" Proceedings of the International Workshop on Planning and Scheduling for Space* (IWPSS 2004). Darmstadt, Germany. June 2004.

[12] Bachelder, A., Hall, J., Jones, J., Kerzhanovich, V. and Yavrouian A. "Titan Aerobot Mission Concepts", *Proceedings of the 4th IAA International Conference on Low-Cost Planetary Missions*, May 2-5, 2000, Laurel, Maryland.

[13] Whalley, M., Freed, M., Harris, R., Takahashi, M., Schulein, G., and Howlett, J. "Design, Integration, and Flight Test Results for an Autonomous Surveillance Helicopter." *Proceedings of the AHS International Specialists' Meeting on Unmanned Rotorcraft*, Jan. 2005. Mesa, AZ.

[14] Freed, M., Harris, R., and Shafto, M. (2004) "Human vs. Autonomous Control of UAV Surveillance." *Proceedings of the American Institute of Aeronautics and Astronautics*, 2004.

[15] "Testable Engineering: The TEAMS Model," URL: http://teamqsi.com/TEAMS.html. Accessed September, 2005.

[16] D'Ortenzio, M. Enomoto, F, Johan, S., "Collaborative Decision Environment for Unmanned Aerial Vehicles," *Proceedings of the AIAA Infotech@Aerospace, Advancing Contemporary Aerospace Technologies and Their Integration Conference, Meeting Papers on Disc (CD - ROM),* AIAA, Arlington, VA, 2005.

[17] Schoenung, S., Wegener, S., Frank, J., Frost, C., Freed, M. And Totah, J. "Intelligent UAV Airborne Science Missions," *AIAA Infotech@Aerospace, Advancing Contemporary Aerospace Technologies and Their Integration Conference, Meeting Papers on Disc (CD - ROM), AIAA, Arlington, VA, 2005.*

[18] Ambrosia, V., Schoenung, S.;,Wegener, S., Enomoto, F, "The 24- Hour UAV Western States Fire Mission: Sensor and Intelligent Management Sysytems," *AIAA Infotech@Aerospace, Advancing Contemporary Aerospace Technologies and Their Integration Conference, Meeting Papers on Disc (CD - ROM), AIAA, Arlington, VA, 2005.*

[19] Freed, M. "Managing Multiple Tasks in Complex, Dynamic Environments." In *Proceedings of the 1998 National Conference on Artificial Intelligence*. Madison, WI.

[20] [ICCM03 paper.]

[21] Freed, M. "Reactive Prioritization." In *Proceedings of the 2nd NASA International Workshop on Planning and Scheduling for Space*. San Francisco, CA.

[22] Allen, J. F. "Maintaining Knowledge about Temporal Intervals," *Communications of the ACM*, Vol. 26, Issue 11, pp. 832-843, 1983.

[24] Apex 2.4 Reference Manual. URL: http://human-factors.arc.nasa.gov/apex/content/referencemanual.html. Accessed September, 2005.

[25] John, B. E., Vera, A. H., Matessa, M., Freed, M., and Remington, R. "Automating CPM-GOMS." In *Proceedings of CHI'02: Conference on Human Factors in Computing Systems*. ACM, New York.

[26] Fitzgerald, W. and Freed M. "Using Delayed Streams to Discern Changing Conditions in Complex Environments: Monitors in Apex 3.0." In *Proceedings of The International Lisp Conference*, June 19-22, 2005, Stanford University.

[27] Bonasso, R.P., Kortenkamp, D., Miller, D. P. and Slack, M. G., "Experiences with an Architecture for Intelligent Reactive Agents" *Proceedings of the International Joint Conference on Artificial Intelligence*, 1995.

[28] Musliner, D., Durfee, E., and Shin, K. "Circa: A Cooperative, Intelligent, Real-time Control Architecture." IEEE Transactions on Systems, Man, and Cybernetics Vol 23, Issue 6, 1993.
[28] Zilberstein, S.. "Resource-Bounded Sensing and Planning in Autonomous Systems," *Autonomous Robots*, Vol. 3, pp. 31-48, 1996.
[29] ARIANE 5: Flight 501 Failure, Report by the Inquiry Board. European Space Agency. Available at URL: http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf, accessed September, 2005.
[30] Freed, M., Harris, R. and Shafto, M.G. (2004) Measuring Performance at UAV-Based Autonomous Surveillance. In *Proceedings of the 2004 Performance Metrics for Intelligent Systems Workshop*, Gaithersburg, MD.

Appendix A: Referenced Reasoning and Control Services

This is a list of the Reasoning and Control Services (RCSs) that Apex provides, and that are mentioned in this paper. This is a non-exhaustive list of RCSs provided by Apex.

- Task Control Services
    - Resource conflict resolution
    - Task state transition control
        - task enablement
        - task interruption
        - task resumption
        - task termination
    - Procedure selection
    - Subtask instantiation
    - Primitive invocation
    - Planner invocation
- Monitoring and Interpretation Services
    - Pattern Matching
    - Resource conflict detection
    - Condition detection (monitoring)
        - constraint checking on individual measurements (value, object, timing)
        - constraint checking on measurement episodes (value, object, timing, statistical trend)
        - constraint checking on estimators
        - constraint checking on logical combinations of measurements and episodes
        - constraint checking on temporal ordering of measurements and episodes
        - value projection and interpolation
    - State variable history management
    - Event history management
- Deliberative Management Services
    - Solver selection
    - Problem classification
    - New procedure generation
- Solvers
    - Modified 2-Opt path planning
    - WAM path planning

Appendix B: Details on monitoring and interpretation services

Monitoring and interpretation services detect task-relevant conditions such as satisfaction of preconditions that make a task eligible to start and violation of runtime requirements that require a task to be interrupted. Checking for a condition may involve monitoring newly arriving data, checking memory (state variable histories) for old data or some combination of the two. Checking a condition may require simply determining whether a particular value exists in a precisely defined interval of state variable history. For instance, the PDL clause `(location +self+ = fuel-bay)` defines a condition in which the agent is

at the fuel-bay. More specifically, the first two symbols refer to a state variable `(location +self+)` representing a history of measurements of the agent's location – i.e. the location of the controlled system. The third and fourth symbol specifies that the condition concerns whether there exists a measurement in the state variable history with the `value fuel-bay`. Since no constraints on what temporal interval within the history should be checked, a default is used. If the condition is expressed within a `waitfor` clause, the default interval starts when the task is created and persists indefinitely (as the task waits for the condition to come about).

Table 1 provides examples of the kinds of constraints on state variables that can be defined in PDL. These include:

- constraints on individual measurements, including on their value, object, or time of occurrence,
- constraints on sets of measurements during an interval (*episodes*), including on their values, timing, aggregate statistical properties and trends
- constraints on estimates of projected future values and interpolated past values, as well as constraints on the estimators themselves
- logical combinations of the above (logical and, or, not),
- constraints on the temporal ordering of episodes, including the full set of Allen predicates[22].

Apex treats event signals in a manner consistent with state variables; events are considered *atomic episodes*; that is, referring to some unknown state variable, and that occur at an instant in time. Thus, conditions for events can be handled in a manner consistent with state variable conditions.

Table 1: Examples of monitor constraint types

Constraint type	*Examples*
Existence constraint in future	`(temperature engine-1 = +setpoint+)`
Existence constraint in past (in general, timestamp constraints)	`(:measurement (temperature engine-1 = +setpoint+)` `:timestamp (< (start-of +this-task+))`
Value constraints	`(temperature engine-1 = +setpoint+ +/- 2)`
Object constraints	`(:measurement (temperature ?engine = +setpoint+)` `:object (tracked-p))`
Future projected value using persistence	`(:measurement (temperature engine-1 = +setpoint+)` `:estimation :persist)`

Past interpolated value using linear regression	`(:measurement (temperature engine-1 = +setpoint+)`
	`:estimation :linear-regression`
	`:timestamp (< (start-of +this-task+)))`

Simple episodic conditions; constrained on value	`(:episode (temperature engine-1)`
	`:quality :no-constraints`
	`:value (> +setpoint+))`

Simple episodic conditions; constrained on aggregate (statistical) values	`(:episode (temperature engine-1)`
	`:quality :no-constraints`
	`:stats (:stddev (> 2.5)))`

Simple episodic conditions; constrained on trends	`(:episode (temperature engine-1)`
	`:quality :no-constraints`
	`:trend (:rate :non-decreasing)))`

Logical constraints and/or/not	`(:and (:or (temperature engine-1 > +setpoint+)`
	`(temperature engine-2 > +setpoint+))`
	`(position flaps = up))`

Temporal constraints on intervals of episodic and measurement conditions (Allen temporal predicates)	`(:in-order (temperature engine-1 > +setpoint+)`
	`(temperature engine-2 > +setpoint+))`
	`(:during (temperature engine-1 > +setpoint+)`
	`(:episode (altitude +self+)`
	`:quality :no-constraints`
	`:trend (:rate :decreasing)))`

Constraints on events	`(:atomic-episode (ready camera-1)` `:timing (< (start-of +this-task+)))`