EUROPA$_2$: Plan Database Services for Planning and Scheduling Applications

Tania Bedrax-Weiss* and **Jeremy Frank** and **Ari Jónsson‡** and **Conor McGann***
Computational Sciences Division
NASA Ames Research Center
Mailstop 269-4
Moffett Field, CA 94035-1000
{bachmann,tania,frank,ajonsson,cmcgann}@email.arc.nasa.gov

Abstract

NASA missions require solving a wide variety of planning and scheduling problems with temporal constraints; simple resources such as robotic arms, communications antennae and cameras; complex replenishable resources such as memory, power and fuel; and complex constraints on geometry, heat and lighting angles. Planners and schedulers that solve these problems are used in ground tools as well as onboard systems. The diversity of planning problems and applications of planners and schedulers precludes a "one-size fits all" solution. However, many of the underlying technologies are common across planning domains and applications. We describe CAPR, a formalism for planning that is general enough to cover a wide variety of planning and scheduling domains of interest to NASA. We then describe EUROPA$_2$, a software framework implementing CAPR. EUROPA$_2$ provides efficient, customizable *Plan Database Services* that enable the integration of CAPR into a wide variety of applications. We describe the design of EUROPA$_2$ from the perspective of both modeling, customization and application integration to different classes of NASA missions.

Introduction

Inspired by NASA's missions that require solving a wide variety of planning and scheduling problems, each of which must be integrated into different operating environments, we set out to formalize and implement a planning framework on which many of these mission scenarios can be built. Our intuition is that many other real-world problems are similar and that such a framework will be widely applicable. The Remote Agent Experiment (RAX) on the Deep Space 1 Spacecraft (Muscettola *et al.* 1998), (Jónsson *et al.* 2000) featured a planner on board a spacecraft that required reasoning about accumulated thrust, spacecraft attitude relative to navigation aids, and the state of hardware resources like cameras. The EO-1 ScienceCraft experiment (Tran *et al.* 2004) is another onboard planner that must reason about onboard memory and CPU resources, communications opportunities to replenish memory, and options for satisfying science goals. Controllers onboard terrestrial Unmanned Autonomous Vehicles (UAVs) such as Rotorcraft (Whalley *et al.* 2003) must reason about the state of communication systems, onboard payloads such as imagers, and how image acquisition constrains intended maneuvers such as banks and climbs, in the face of complex flight dynamics. Autonomy systems (Dias, Lemai, & Muscettola 2003), (Despouys & Ingrand 1999) as well as ground tools (Bresina *et al.* 2003) for robots like the Mars Exploration Rovers (MER) require reasoning about thermal models, available power and remaining memory, as well as the location of the rover relative to intended science targets and how to choose from among available science operations. Image Processing planning (Golden *et al.* 2003) requires reasoning about feasible image manipulation operations, available web services, as well as the state of underlying computer file systems, including the location of inputs and outputs of processing operations.

The diversity of planning problems and applications of planners and schedulers precludes a "one-size fits all" solution. Different planning paradigms apply more naturally to different planning problems, and different applications require different planning services. For example, planetary rover domains require one form of path planning, UAVs require quite different forms of path planning, while satellite domains such as EO-1 do not require path planning at all. Path planning generally requires reasoning about concepts that are immutable with respect to time, and so does image processing. Although domains such as EO-1, MER, and RAX require reasoning with resources, EO-1 and MER feature onboard memory resources, while the RAX does not. In either of these cases, reasoning about time is important. Furthermore, in onboard systems such as spacecraft, UAVs and rovers, planner response time may preclude expensive algorithms that guarantee optimality. Additionally, some applications require that the planner provide incomplete solutions, such as those where the planner interfaces with an intelligent executive that is able to "fill in the blanks". Human operators or other autonomous sub-systems may look at plans, and request changes or explanations, ultimately leading to new planning problems.

Despite the considerable diversity of planning problem classes, planners and applications, there is considerable commonality among planning and scheduling problems,

*QSS Group, Inc.
†Authors listed in alphabetical order.
‡USRA-RIACS

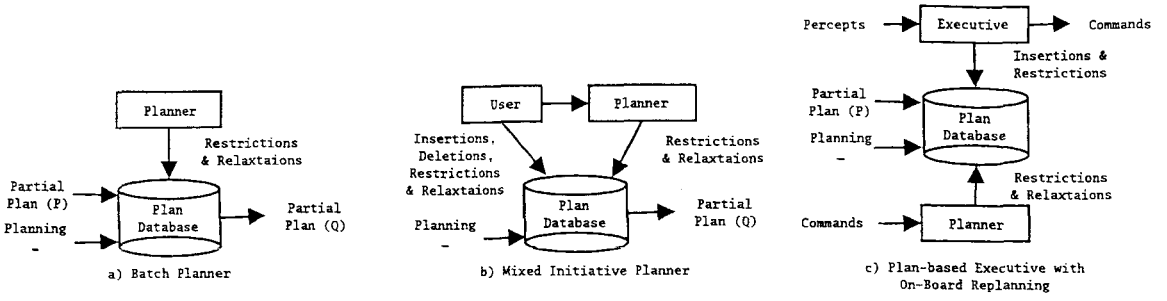

Figure 1: Sample Plan Database Applications

solvers and applications. This commonality can be aggregated into a set of plan services that we call the *Plan Database* that are provided to build such applications. Consider the scenarios illustrated in Figure 1. The first is an application of automated planning where the input planning problem is solved by a *Planner* to produce an acceptable partial plan. The role of the *Planner* is to perform the search steps for resolving flaws. Thus it interacts with a partial plan by imposing and retracting restrictions. All operations are made on the *Plan Database* which stores the partial plan. The second is an application of automated planning in concert with a *User*. The User may introduce goals into a plan, and change or undo decisions previously made by a Planner. Additionally, a User may employ a Planner to work on the current partial plan. In this case changes are also made in response to queries and operations on the *Plan Database*. In the last figure, planning technology is deployed for plan execution. A partial plan may be used by an *Executive* for execution. In such a scenario, the partial plan is updated throughout execution. The Executive may employ incomplete search to refine the partial plan as it goes. A Planner may be employed to repair a plan or develop a refinement of the plan as the mission progresses. In each of the cases described, *clients* (i.e. Planner, User, Executive) leverage the services of a common *server*, the *Plan Database*.

We have created a robust formal framework called Constraint Planning with Resources (CAPR) that supports many commonly used representational primitives and reasoning engines. We describe this formalism in the next section of the paper. This formal framework provides the underpinnings for the Plan Database, called the Extensible Universal Remote Operations Architecture (EUROPA$_2$). This idea is similar to the approach taken by the CLARATy robotics control architecture (Nesnas *et al.* 2003) or MDS (Dvorak *et al.* 2000), as well as constraint reasoning systems such as ILOG (ILOG 1996).

Applications will require customization of the Plan Database to support only those primitives needed by the domain (e.g. time, resources), and to implement an appropriate planner (e.g. an optimizing planner versus one with real-time guarantees). We describe how to build domain models for EUROPA$_2$ as well as how to build custom planners. In the final sections of the paper, we discuss related work, and conclude with a discussion of our future plans.

Constraint-Based Planning with Resources

In this section we describe Constraint Based Planning with Resources (CAPR). CAPR is a modification of Constraint-Based Attribute and Interval Planning (CAIP) (Frank & Jónsson 2003), a formalism that employs variables and constraints as first-class objects to describe complex planning domains. CAPR relaxes some of the more restrictive assumptions made in CAIP, resulting in a more generally applicable formalism. In particular, we include general resources as first-class citizens in the planning formalism, and separate subgoaling and causal models from the resource model. We will show later that we lose none of the representational power of CAIP by having made these changes.

We first describe the formalism in grounded terms, in which all primitives are predicates. We then provide a more easily managed formalism using constraints and variables as primitives.

Grounded case

A *token* is a logical statement of the form $\texttt{holds}(s, e, p(a_1, ..., a_k))$ where $t_s < t_e$ are start and end times, p is a predicate symbol and $a_1, ..., a_k$ are parameter values. Tokens generalize actions and state, and merely assert that some property of interest is true for a period of time.

A *resource* R is defined by a tuple (i_R, l_R, L_R) where i is the initial level, l is the minimum level, L is the maximum.

A *transaction* is a numerical change in a resource over a specified interval. It is defined as a tuple (R, s, e, δ) where R is a resource, $s \leq e$ are times, and δ is a function mapping timepoints t, $t \in [t_s, t_e]$, to numerical values.

An *instantaneous transaction* is a transaction where $t_s = t_e$ and is referred to as (R, t, δ) .

A *configuration rule* is an implication of the form $T \Rightarrow C_1 \vee C_2 \vee \cdots \vee C_n$ where T is a token and each C_i is a conjunction of the form $S_{i,1} \wedge \cdots \wedge S_{i,k_i}$ where each $S_{i,j}$ is either a token or a transaction.

Definition 1 *A* planning domain $\mathcal{D}$ *is a tuple* $(\mathcal{T}, \mathcal{R}, \mathcal{C})$, *where* $\mathcal{T}$ *is a set of tokens,* $\mathcal{R}$ *is a set of resources, and* $\mathcal{C}$ *is a set of configuration rules.*

Definition 2 *A* resource profile *for a given plan* P *and resource* (i_R, l_R, L_R) *from the domain for that plan is a function* $\lambda_R(t)$ *defined as follows:*

- *We first define a cumulative impact function Δ_i for each transaction T_i in P as follows:*
  - *If T_i is a non-instantaneous transaction , define Δ_i as the integral of δ, defined as $\Delta_i(t) = 0$ for $t < t_s$, $\Delta_i(t) = \int_{\tau=t_s}^{t} \delta(\tau)$ for $t \in [t_s, t_e]$ and $\Delta_i(t) = \int_{\tau=t_s}^{t_e} \delta(\tau)$ for $t > t_e$.*
  - *If T_i is a instantaneous transaction , define $\Delta_i(t) = 0$ if $t < t_s$, and $\Delta_i(t) = \delta(t)$ if $t \geq t_s$.*
- *Then, for each time point t, $\lambda_R(t) = \sum_i \delta_i(t)$.*

A resource profile $\lambda_R(t)$ for a resource (i_R, l_R, L_R) and plan P is *valid* if $l_R \leq \lambda_R(t) \leq L_R$ for all timepoints t.

A *partial plan* is a set of tokens along with the applicable transactions defined by the domain rules.

A partial plan Q is an *extension* of a partial plan P if each token in P can be mapped to a matching token in Q.

Definition 3 *A partial plan P is* valid *if:*

- *for each token T in P, and for each configuration rule $T \Rightarrow C_1 \vee \cdots \vee C_n$, there exists a $j \in [1, n]$ such that where $C_j = S_{i,1} \wedge \cdots \wedge S_{i,k_i}$, each of the tokens and transactions $S_{i,1}, \ldots, S_{i,k_i}$ are in P.*
- *the resource profile for every resource is valid*

A *planning problem* is a pair, $(\mathcal{D}, P)$ where $\mathcal{D}$ is a planning domain and P is a partial plan. A *solution* to the planning problem is a plan Q that is a valid extension of P.

Lifted case

The grounded formalism is inconvenient since it may require large numbers of token descriptions and rules. It is more effective to compress these definitions by using variables and constraints as the primitive elements of the planning domain descriptions.

A *domain* is a list of primitive values. A *predicate definition* is a tuple $(p, D_1, ..., D_k)$ consists of a predicate p and a (possibly empty) set of domains, which define the number of arguments and the argument domains for the predicate.

A *resource definition*, like before, is a tuple (i_R, l_R, L_R) where i is the initial level, l is the minimum level and L is the maximum.

A token specifies a predicate instantiation holding over a period of time. Formally, a *token* is a tuple $(s, e, p, a_1, ..., a_k)$ where s and e are temporal variables, and each a_i is a variable whose domain is restricted to D_i. (Note that a duration variable d can be defined for convenience, but is not necessary.) We distinguish the domain of a variable a_i in a token as $\texttt{domain}(a_i)$, as opposed to a domain used in a predicate definition.

A *transaction* is a defined by (R, s, e, δ) as before, except that R, s and e are variables. Instantaneous transactions enforce the constraint $s = e$.

A *compatibility* is a way to represent large collections of configuration rules compactly. It is an implication of the form $H \Rightarrow B_1 \vee B_2 \vee \cdots \vee = B_n$. The head H is a tuple $(p, E_1, ..., E_k)$, where p appears in a planning domain predicate definition $(p, D1, \ldots, D_n)$ such that $E_i \subseteq D_i$. Each B_i is a conjunction of the form $S_{i,1} \wedge \cdots \wedge S_{i,k_i}$ where each $S_{i,j}$ is of the form: $G_{i,j}; C_{i,j}$ where $G_{i,j}$ is a predicate or a transaction, and $C_{i,j}$ is a set of constraints relating variables in the head predicate and $G_{i,j}$. A token $(s, e, p, a_1, ..., a_k)$ *matches* a compatibility head $(q, E_1, ..., E_k)$ if $p = q$ and $\forall i, \texttt{domain}(a_i) \subseteq E_i$.

A *planning domain* is a tuple $(\mathcal{P}, \mathcal{R}, \mathcal{C})$ where $\mathcal{P}$ is a set of predicate definitions, $\mathcal{R}$ is a set of resource definitions, and $\mathcal{C}$ is a set of compatibilities.

A *resource envelope* for a given plan P and resource $R = (i_R, l_R, L_R)$ is a pair of functions $L_{max,R}(t)$ and $L_{min,R}(t)$ which are defined as follows: Let $Q_1, Q_2, \ldots$ be the set of all grounded extensions of P. Let $\lambda^i_R(t)$ be the resource profile for Q_i. Then $L_{max,R}(t) = \max_i \lambda^i_R(t)$ and $L_{min,R}(t) = \min_i \lambda^i_R(t)$. A resource envelope is *valid* if $l_R \leq L_{min,R}(t) \leq L_R$ and $l_R \leq L_{max,R}(t) \leq L_R$ for all times t. A resource envelope is , *violated* if either $L_{max,R}(t) < l_R$ or $L_{min,R}(t) > L_R$ for some t. A resource envelope is *undetermined* if it is neither valid nor violated.

A constraint c is a relation among the values of a set of variables $a_i ... a_k$; that is, $\mathcal{L} \subset \texttt{domain}(a_1) \times ... \times \texttt{domain}(a_k)$. A constraint c is *satisfied* if all possible instantiations of its variables yield assignments in the relation $\mathcal{L}$. A constraint c is *violated* if no instantiation of its variables yields an assignment within the relation $\mathcal{L}$. Finally, a constraint is *undetermined* if it is neither satisfied nor violated.

A *partial plan* is a set of tokens and a set of constraints. Each token in a partial plan is either *supported* or *unsupported*. A token T is supported if for every compatibility where the head matches with T, the compatibility has at least one disjunct B_i such that for each conjunct $G_{i,j}; C_{i,j}$ in B_i, the plan contains a token that matches $G_{i,j}$ and has all corresponding constraints in $C_{i,j}$. Any token that is not supported is unsupported. Finally, a given partial plan P, defines a set of resource transactions, and associated resource envelopes.

A partial plan P is *complete* if all tokens are supported. A partial plan P is *valid* if the resulting resource envelopes are valid, and all constraints in P are satisfied.

A *planning problem* is a planning domain and a partial plan P from that domain. A *solution* to the planning problem is a complete and valid plan Q that is an extension of P.

Decision Model and Completeness results

We next describe the flaw mechanisms and the associated search path options. In backwards chaining, unsatisfied preconditions are flaws that must be resolved before achieving an complete plan. In POCL planning, the flaws are open conditions and unresolved threats. In CAPR, flaws are either undetermined constraints, undetermined resources, or unsupported tokens. As we will see below, flaw resolution for all three of these cases is accomplished by constraining the domain values of variables.

Undetermined constraints: Suppose we have a partial plan P with a variable v in a constraint c that is undetermined. Normally, unassigned variables are simply assigned single values until constraints are known to be satisfied. However, it is possible to proceed by imposing constraints that restrict variables' values.

Undetermined resources: Suppose we have a partial plan P with a resource that is undetermined. In most cases it is too expensive to calculate $L_{max,R}(t)$ and $L_{min,R}(t)$, because it would require calculating all of the grounded extensions Q_i. Thus we must bound above $L_{max,R}(t)$ and bound below $L_{min,R}(t)$ to determine validity. When all transactions are grounded we can determine $L_{max,R}(t)$ and $L_{min,R}(t)$; for this reason, flaws on resources are usually satisfied by assigning transaction timepoint variables. Suppose the problem is such that no incomplete token decisions will ever arise as flaws are resolved. In this case, we are left with a *scheduling problem.* If we further restrict ourselves to the case of scheduling instantaneous transactions, we can use techniques such as those described in (Frank 2004; Muscettola 2002) to tightly bound $L_{max,R}(t)$ and $L_{min,R}(t)$. In some circumstances, partial orders of transactions are sufficient to guarantee that the resource is provably valid. For these cases, flaw resolution can be accomplished by only ordering transaction timepoints.

Unsupported tokens: Finally, suppose we have a partial plan P with a token $T = (s, e, p, a_1, ..., a_k)$ that is unsupported. There is at least one rule whose head unifies with (matches) T. For each such rule, one of the disjuncts B_i must be chosen in order to satisfy the rule. This can be thought of as a value choice for a variable. Each disjunct consists of a conjunct $G_{i,j}; C_{i,j}$ where $G_{i,j}$ is a predicate description or transaction. If $G_{i,j}$ is a transaction, a resource must be chosen for the transaction; this too is a variable choice. If $G_{i,j}$ is a token, then let $\mathcal{V}$ be the set of tokens that can be unified with $G_{i,j}$, along with one extra element, $\top$, representing the use of a new token. Then, the decision to be made is which element of $\mathcal{V}$ to select. Once again, this can be viewed as a variable choice. Note that only if $\top$ is chosen, resulting in a new token, will any new compatibilities apply to tokens in the plan P. However, if $G_{i,j}$ is unified with $V \in \mathcal{V}$, all the constraints in $C_{i,j}$ are added to constrain the variables in V and T. These constraints generalize causal links in the same manner as CAIP.

Completeness results: We are now ready to show that this decision model is sufficient for solving planning problems in CAPR. As was true in the CAIP framework (Frank & Jónsson 2003), there may be solutions to a planning problem that are not reachable given the domain description and the decision model. However, we can still prove that there is a plan that is a complete and valid extension of the domain description and decision model such that the unreachable plan is an extension of this plan. This situation arises because there is nothing in the formalism to prevent adding arbitrary tokens that don't have compatibilities associated with them.

Theorem 1 *Given a finite planning domain $(\mathcal{P}, \mathcal{R}, \mathcal{C})$ and a finite length partial plan P. Assume that Q is a complete and valid finite length extension of P. Then, there exists a plan R, that is a complete and valid extension of P such that a sequence of flaw resolutions transforms P into R, and Q is an extension of R.*

Proof 1 *As in (Frank & Jónsson 2003), we will use Q as a "heuristic" to describe how to transform P into Q. While applicable:*

- *If a token T of P is unsupported, there is a supported token V in Q that matches T; use this token to satisfy T, either by choosing a disjunct B_i, by satisfying a conjunct $S_{i,j}; C_{i,j}$ with an existing matching token in P, or by adding a new token to P.*
- *If a variable v is unassigned, there is a matching variable w in Q; use this variable to assign the value of v. Note that this covers the case of deciding which available resource a transaction is assigned to.*
- *If a constraint among variables in P has not been imposed, use Q to impose that constraint. Note that this covers the case of ordering timepoints.*

Since Q is finite and P, at each stage, is a subset of Q, the process halts with a complete plan R. And, since the set of constraints in P, at each stage, are a subset of those in Q, constraint validity is obvious. The only remaining part is to show that all resources are valid in P. First, it is easy to see that a resource in P cannot be violated, as Q is an extension of P and the profile is defined based on all extensions. Second, the resource cannot be neither violated nor valid, as that will give rise to flaws and the process does not halt until there are no other flaws. So, the resource envelopes must also be valid. Thus, R is a complete valid extension of P, and is a subset of Q; thus any tokens, constraints or transactions in Q can be added to R with impunity.

EUROPA$_2$

In order to successfully deploy CAPR in many different contexts, we adopt the strategy of providing a *Plan Database* motivated by the CAPR formalism. The Plan Database provides services that support description of planning domains, allow implementation of a wide variety of planners and schedulers, as well as provide information about the plans as well as the planning process to applications. These services include:

- Domain modeling: for describing planning domains
- Partial plan representation: for maintaining partial plans
- Flaw generation: for generating flaws from a partial plan
- Flaw resolution: for resolving flaws in a partial plan
- Plan assessment: for determining plan completeness or violations
- Constraint propagation: for propagating the consequences of constraints

To meet the needs of missions and research projects, the design of the Plan Database must meet the following critical design goals:

- Efficiency - ensure low latency for operations and queries.
- Flexibility - ensure services can be selected and flexibly integrated.
- Extensibility - ensure services can be enhanced to meet the needs of research or mission applications.

We have implemented EUROPA$_2$, a Plan Database that meets these requirements. To illustrate EUROPA$_2$ we present a planning domain loosely based on the MER mission. We assume the application in question is one of producing daily activity plans for operation of a planetary surface robot named *Rover*. *Rover* is a mobile robot equiped with a range of instruments to sample and study a geological site. A *Rover* processes plans for taking rock samples in various locations within a given survey area. It has on board a battery, and can replenish its energy levels using solar power.

Planning Domain Descriptions with NDDL

Planning domain descriptions for EUROPA$_2$ are written in New Domain Description Language (NDDL). In this section we will describe NDDL and show how the syntax translates to the CAPR formalism.

Rooted in the formal framework of CAPR, NDDL provides an object-oriented syntax and semantics that makes it convenient to express sophisticated relationships among elements of a partial plan. There are a number of additional capabilities in NDDL which offer greater convenience and or efficiency for the modeler.

Predicates A predicate in CAPR defined as $(p, D_1, ..., D_k)$ is directly described in NDDL. For example, a *Rover* might be at a *Location*, or it might be moving from one location to another. The predicate *At* can be introduced with:

```
predicate At{Rover r; Location l;}
```

where r and l refer to the set of all rovers and the set of all locations respectively. Similarly we can introduce the predicate *Going*:

```
predicate Going{Rover r;
                Location from;
                Location to;}
```

Rover and *Location* are user-defined types which may be expressed using enumeration:

```
enum Rover {spirit, opportunity}
```

or through the more expressive use of an abstract data type, or *class*:

```
class Rover {}
class Location {
  int x;
  int y;
  Location(int _x, int _y){
    x = _x;
    y = _y;
  }
}
```

Thus, class describes an unchanging object. Instances of classes, i.e. objects, may be introduced thus:

```
Rover spirit = new Rover();
Rover opportunity = new Rover();
Location rock = new Location(1, 1);
Location hill = new Location(2, 3);
Location lander = new Location(5, 8);
Location martian-city = new Location(8, 6);
```

For convenience, predicates may be defined directly on a class. Predicates introduce time-varying properties of a class. The set of instances of that class are implicitly a parameter of the predicate, and are accessed through the built-in variable *object*. Thus we may concisely restate our predicate definitions by augmenting the *Rover* class:

```
class Rover {
  predicate At{Location l;}
  predicate Going{Location from;
                  Location to;}
}
```

Compatibilities Suppose that *Rover* is not permitted to go to the same location it is leaving. Furthermore, suppose that every *Going* must be followed by an *At* and vice versa. To express these domain rules, we introduce a *compatibility* for each predicate. Recall that a token is defined in CAPR as $(s, e, p, a_1, ..., a_k)$. The compatibility for *At* given below shows the two *Going* subgoals with constraints imposed on their predicate parameters (including the previously described implicit *object* variable) and its *start* and *end* variables.

```
Rover::At{
  // Require a Going token on same
  // object which succeeds this token
  subgoal(Going g0);
  eq(g0.start, end); // Equate timepoints
  eq(g0.from, l); // Equate parameters
  eq(g0.object, object);
  // Require a Going token on same
  // object which precedes this token
  subgoal(Going g1);
  eq(g1.end, start); // New constraint
  eq(g1.to, l); // New constraint
  eq(g1.object, object);
}
```

NDDL directly supports specifying temporal constraints with Allen relations augmented with metric time. We use the Allen relations directly as shorthand for creating a subgoal token with the associated temporal constraints. Furthermore, we can use the `object` variable to specify the constraint that the `At` token must be on the same object as the `Going` token. We thus express the compatibilities for *Going* more concisely as follows:

```
Rover::Going{
  neq(to, from); // to != from
  meets(object.At a0);
  eq(a0.l, to);
  met_by(object.At a1);
  eq(a1.l, from);
}
```

Suppose the *Rover* could either go to another location or stay at the current location and take a panoramic image. In NDDL we model this by using a disjunctive rule where we explicitly create a boolean variable to represent the disjunction.

```
Rover::At{
  // disjunctive rule for successor
  // token:  false implies Going, true
  // implies TakeImg
  bool next;
  if (next==false) {
     meets(object.Going g0);
```

```
    eq(g0.from, l);
  }
  if (next==true) {
  meets(object.TakeImg i0);
  }
...
}
```

Resources and Transactions To illustrate the use of resources in NDDL, we introduce a battery which stores energy produced from solar panels and allows energy to be consumed by rover activities.

```
class Rover {
  ...
  Resource battery;
  Rover(){
    ...
    battery = new Battery(10, 3, 30);
  }
}
```

Now declare a predicate for power generation:

```
predicate generatePower{Resource r;
                        float rate;}
```

and define a rule linking it to transactions on a resource. Note that the current EUROPA$_2$ implementation is limited to handling instantaneous transactions. Consequently, transactions are typically defined as occurring at the start or end of tokens. Instantaneous transactions in CAPR are defined by (R, t, δ) and are identical in NDDL:

```
generatePower{
  // produce transaction at the end
  ends(r.transaction tx);
  // relation to derive instantaneuos
  // change from rate and duration
  calcProduction(tx.quantity,
                 rate, start, end);
}
```

Finally, the compatibility for *Going* can be augmented with a consumption transaction on the battery where the quantity is based on the distance travelled:

```
...
subgoal(object.battery.transaction tx);
calcConsumption(tx.quantity, from, to);
// Consume at the beginning
eq(tx.start, start);
...
```

A common special case of resources is the unary resource specifying a mutual exclusion between states and actions that cannot occur simultaneously. In the *Rover* example, *At* and *Going* tokens are temporally mutually exclusive for any given *Rover* instance. To accomplish this, we embed a *Resource* into the *Rover* class in accordance with the definition (i_R, l_R, L_R):

```
class Rover {
  ...
  Resource mutex;
  Rover(){
    mutex = new Resource(1, 0, 0);
  }
}
```

Now, we append an appropriate transaction requirements to our existing compatibilities.

```
...
// Consume at the beginnning
subgoal(Resource.transaction tx0);
eq(tx0.object, object.mutex);
eq(tx0.time, start);
eq(tx0.quantity, -1);
// Produce at the end
subgoal(Resource.transaction tx1);
eq(tx1.object, object.mutex);
eq(tx1.time, end);
eq(tx1.quantity, 1);
...
```

Timelines In CAIP, Timelines were constructs used to ensure that a set of predicates were mutually exclusive, as well as ensuring that one of these predicates held at any time in valid plans. In CAPR, and subsequently, EUROPA$_2$, timelines are no longer first class members of the paradigm. However, the notion of ensuring mutual exclusion among predicates is very common. Therefore, it's useful to have a convenient short-hand for defining classes, where all their tokens are mutually exclusive. This is done by declaring a class as an extension of a special construct called a *Timeline*. Using this construct is exactly the same as defining mutual exclusion unary resources and appropriate resource transactions, but is more concise and can be implemented efficiently. In our example, the use of *Timeline* gives a more concise model:

```
class Rover extends Timeline {
  predicate At{Location l;}
  predicate Going{Location from;
                  Location to;}
  Resource battery;
  Rover(){
    battery = new Battery(10, 3, 30);
  }
}
```

Static Objects Other useful features offered by NDDL are local compatibility variables, and the use of classes to capture information that is static over time. These features prove useful in the *Rover* planning domain where a further restriction is imposed such that only some paths in the survey area are traversable. The abstract data type for the set of paths can be specified as:

```
class Path {
  Location loc1;
  Location loc2;
  Path(Location _l1, Location _l2){
    loc1 = _l1;
    loc2 = _l2;
  }
}
```

and the set can be populated with instances using object allocation e.g.:

```
Path p1 = new Path(rock, hill);
Path p2 = new Path(hill, lander);
Path p3 = new Path(martian-city, lander);
```

An additional rule can be introduced for the *Going* predicate to enforce the path existence requirement:

```
Rover::Going{
  Path p :  {
```

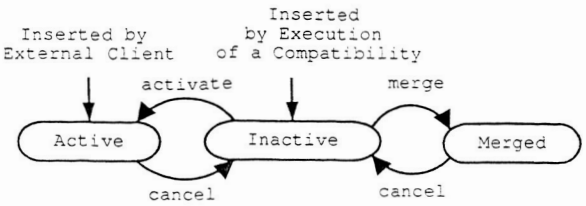

Figure 2: Token State Transition Diagram

```
    eq(p.loc1, from);
    eq(p.loc2, to);
  }
}
```

The variable p is comparable to a predicate parameter variable, though it is only introduced locally in a rule, and need not be grounded in a complete partial plan. The initial values for p will be *p1, p2,* and *p3*. These values will be filtered through constraint propagation. Should there be no path, the domain of p will be empty and a violation will occur.

Partial Plans in EUROPA$_2$

In this section we discuss the representation and manipulation of partial plans in EUROPA$_2$.

Tokens and Open Conditions A partial plan for the rover planning domain is created with the following statement:

```
goal(Rover.Going G);
```

This introduces a token G for the predicate *Going* defined on the class *Rover*. The results is the partial plan $p = \{\{G\},\{\}\}$. All tokens in a partial plan are represented as *Active Tokens* in a EUROPA$_2$ plan database. All *Open Conditions* can be inferred from the partial plan and the model. Open Conditions are represented in a EUROPA$_2$ plan database as *Inactive Tokens*. Figure 2 illustrates the states and transitions of tokens in EUROPA$_2$. As is the case with G, a token is *Active* immediately when introduced by an actor external to the plan database. Alternatively, a token is initially *Inactive* when introduced by a compatibility. As described in the previous section on the decision model, an *Inactive Token* must be resolved by either *merging* it with an existing *Active Token* (i.e. choosing a resolver from the set of tokens in the plan $\mathcal{V}$) or by inserting it into the partial plan via *activation* (i.e. using the resolver $\top$).

With G the following variables are introduced to the Plan Database. The *default variables* of G are introduced with all tokens:

- *start* - the start time for the token. In this example the domain is [-inf +inf].
- *end* - the end time for the token. In this example, the domain is [-inf +inf].
- *duration* - the duration of the token. This does not add any new information to the definition of a token since it can be derived from the *start* and *end* variables but it proves convenient. In this example, the domain is [1 +inf].

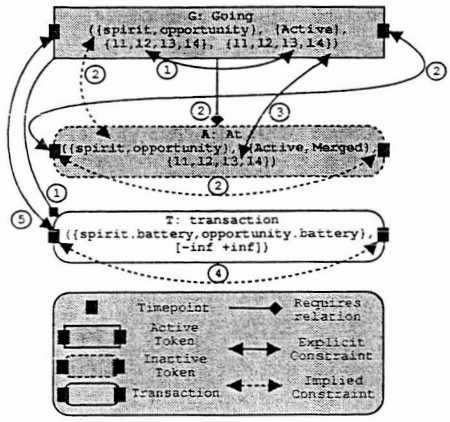

Figure 3: Plan database elements for partial plan $\{\{G\}\{\}\}$

- *object* - the implied variable arising from the definition of the predicate on a class or a transaction on a resource. In this example, the domain is populated with all instances of the *Rover* class i.e. {spirit, opportunity}.
- *state* - annotation for the state of a token. An *Inactive Token* has a domain of {Active, Merged}. A *Merged Token* has the singleton domain *Merged*. The domain for G is a singleton {Active}. The operations to *activate* or *merge* a token constrains this variable.

The *parameter variables* introduced depend on the predicate description of the token. In this case, since G is an instance of the `Going` predicate, we introduce the following:

- *from* - the location the rover is leaving from. In this example the domain is populated with all instances of the *Location* class i.e. {rock,hill,lander,martian-city}.
- *to* - the location the rover is going to. In this example the domains are identical.

Constraints Further requirements can be imposed in the initial partial plan. For example, *spirit* must be at location *rock* at time *0*:

```
// Introduce token A
goal(Rover.At A);
// Constrain location variable
eq(A.l, rock); // c0
// Constrain object variable
eq(A.object, spirit); // c1
// Constrain start <= 0 <= end
leq(A.start, 0); // c2
leq(0, A.end); // c3
```

Taken together, this partial plan, p, is given by the tuple ({G,A}, {c0, c1, c2, c3}).

Inference with Compatibilities Compatibilities are only matched with *active tokens*. A simplified version of a previouosly described compatibility for *Rover::Going* is listed

below:

```
0.  Rover::Going{
1.    neq(to, from); // to != from
2.    meets(object.At A);
3.    eq(A.1, to);
4.    subgoal(object.battery.transaction T);
5.    eq(T.start, start);
6.  }
```

The head of the above compatibility is matched immediately upon introduction of *G* to the partial plan. Execution of the body yields an *open condition A*, a set of *constraints* and a single transaction *T*. These elements, and the relations between them, are illustrated in Figure 3. The relations are annotated with line numbers indicating where they arise from explicit declarations in the compatibility. Line 1 produces a constraint among the parameter variables of *G*. Line 2 introduces the open condition *A*. It also imposes an *equality* constraint between the *object* variables of *G* and *A*. Line 3 equates the parameter variables *A.l* and *G.to*. Line 4 requires a new transaction *T* in the database. EUROPA$_2$ does not currently support interval transactions, so we also generate an implicit constraint equating *T.start* and *T.end*. Finally, Line 5 equates the start times of *G* and *T*. Note that disjunctive compatibilities are modeled by variables, so these variables would be introduced as flaws when matching a compatibility to an *Active* token, and only when these variables are decided do we introduce the corresponding tokens and constraints.

Flaw Generation and Resolution

CAPR identifies flaws as undetermined resources, unsupported tokens, and undetermined constraints. In EUROPA$_2$ these translate to resource, token, and variable flaws respectively. Queries and events are provided so that clients can readily access flaws from the Plan Database. Events provide immediate access to changes within the Plan Database, but require clients to subscribe in order to receive the updates. For example, when an *Inactive* token is inserted into the plan database through execution of a compatibility, a message to that effect is posted to any registered clients. Similarly, as variables are introduced, restricted or relaxed, clients may observe these events and synchronize their flaw state accordingly. Furthermore, events are raised as resource profiles become valid or undetermined. Alternatively, clients may simply query the database for the current set of all unbound variables, open conditions and undetermined resources.

The following methods of resolution are provided in EUROPA$_2$ for each category of flaw:

- *Resource Flaw* - a transaction may be constrained to a resource by assigning or constraining its *object* variable. Alternatively, transactions may be ordered with respect to other transactions on the resource by posting constraints on timepoints.
- *Token Flaw* - inactive tokens required by each unsupported token must be *activated* or *merged*. The former is simply a restriction to the *state* variable of the inactive token to the value *Active*. The latter can be accomplished with a restriction to the *state* variable domain to *Merged* and the posting of equality constraints between the matched variables of the inactive token and the target active token with which it is to be unified. However, a more efficient operation is provided which eliminates the need for equality constraints. This provides significant performance advantages as it reduces the growth rate of the resulting constraint network.
- *Variable Flaw* - unbound variables are resolved by assigning values directly or restricting values with constraints.

Plan Assessment

Some applications may have different models of interaction with EUROPA$_2$ and will want to impose relaxations on the set of flaws that should be resolved by the planner. For example, imagine a multi-agent system where each planning agent shares a single model, yet each is specialized to resolve flaws only in a sub-domain of expertise. Each planning agent would inspect the shared database and work on those flaws it knows how to resolve. Each planning agent would be done planning when it finished resolving all the flaws it needs to resolve. EUROPA$_2$ provides a flexible decision management framework to filter the set of flaws that need to be resolved for a partial plan to be complete. Semantically, these operations amount to a *relaxation* of the strict interpretation of the set of flaws in a plan. The filtering criteria allow clients to indicate:

- temporal restrictions - all flaws outside a given planning horizon are excluded.
- predicate restrictions - all flaws derived from a given set of predicates are excluded.
- variable restrictions - variable flaws on a given set of dynamic and/or infinte variables are excluded.
- custom restrictions - specialized filter conditions may be developed and integrated as needed by the client.

Constraint Propagation

EUROPA$_2$ is built upon the constraint propagation infrastructure illustrated in Figure 4. The model statement:

```
calcConsumption(tx.quantity, from, to);
```

introduces a *Constraint* with the *ConstrainedVariables T.quantity, from,* and *to*. Each constrained variable has a domain which is propagated. A change in a domain triggers a message in the variable, which is passed on to the *ConstraintEngine*. Each constraint is registered with a *Propagator* allowing customized propagation strategies for different constraints. This framework allows for sepcialized domains, constraints, variables and propagators to be integrated in an open and flexible manner. The framework borrows heavily from the design of the CHOCO kernel (Laburthe & the OCRE Research Group 2001). EUROPA$_2$ provides a library of useful constraints together with a default propagator which delegates constraint enforcement to each individual constraint. It also includes a resource propagator which propagates transaction loads on resources, and a temporal propagator which propagates temporal constraints using a simple temporal network.

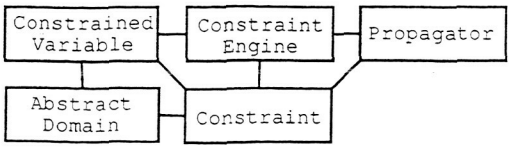

Figure 4: Constraint Propagation Framework

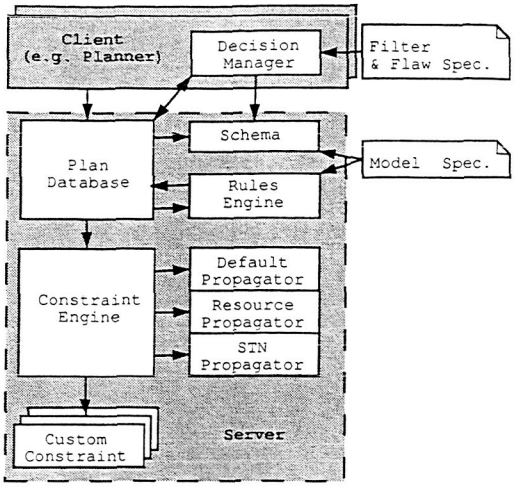

Figure 5: System Diagram - preliminary

EUROPA$_2$ Architecture

We now describe the overall EUROPA$_2$ System Architecture and discuss how it accomplishes design goals described earlier.

Figure 5 describes the internals of the EUROPA$_2$ Plan Database operating as a server to one or more clients. The server is an assembly of EUROPA$_2$ components integrated for the needs of the particular application. The *Plan Database* provides a facade for interacting with the server at the abstraction level of primitives in CAPR i.e. tokens, transactions, constraints, resources, variables. The *Constraint Engine* and related components are utilized to propagate relations among variables and detect violations. Standard and customized constraints and propagators can be freely integrated or omitted. The *Rules Engine* is triggered by changes in the partial plan i.e. token activation and variable binding. The *Schema* is the in-memory store for the domain model. It is used by the plan database to enforce type restrictions and by the rules engine to match and execute compatibilities. EUROPA$_2$ includes a chronological backtracking planner as a standard client component, though many applications develop their own clients. The *Decision Manager* uses a flaw filter specification to manage the set of flaws for a client.

Customizability EUROPA$_2$ is highly customizable. If a problem does not require resources, support for resources may be ommitted. If a problem does not require compatibilities (e.g. a scheduling problem), the rules engine can be omitted. If temporal constraints are not important in a problem, the temporal propagator may be removed and/or replaced with the default propagator. Only required constraints need to be registered. This form of customization is useful as it allows systems to avoid incurring costs for components that are not required. EUROPA$_2$ also provides a language to customize the system for new domain models. Furthermore, heuristic and flaw specifications are also provided. Finally, an open API ensures flexbility in how EUROPA$_2$ is integrated.

Extensibility EUROPA$_2$ is highly extensible. As new problems are encountered, or new algorithms are developed, there are many ways to integrate new capabilities as specialized components e.g. constraints, propagators, resources. This is essential for success in research and mission deployments.

Speed EUROPA$_2$ has produced significant gains in speed over EUROPA. The primary contributors to the improvement arise from:

- fast interfaces & specialized implementations. The ability to tune implementations using inheritance provides speed improvemnts in key areas such as operations on domains.
- efficient merging. Resolving open conditions by merging is an important operation governing the efficiency of the system. EUROPA$_2$ accomplishes this with an algorithm that avoids redundant constraints arising in the plan database.
- incremental relaxation. When relaxing a variable (e.g. retracting a decision), EUROPA$_2$ uses localized propagation to relax reachable variables in the constraint graph.
- direct support for static facts. EUROPA$_2$ uses objects to capture static facts. We provide a means to naturally reference or require objects through variables and the pattern for existential quanitification. EUROPA used single predicate timelines to capture this information, incurring a high overhead and compounding the problems of inefficient merging.

Future Work

We have presented a formalization of constraint-based planning with resources and described EUROPA$_2$ a framework that implements the formulation. The current implementation of EUROPA$_2$ is being used by the Intelligent Systems Program to demonstrate advanced robotic capabilities in the field. We have plans to make this software available for use in research and mission deployments. We are currently working on many extensions. On the theoretical side, we plan to develop domain independent heuristics for resource-cognizant planners. The main challenge is the identification of useful heuristics and the translation of static CSP heuristics into a dynamic CSP setting. We also plan to work on obtaining soundness and completeness results for different subgoaling configurations. We know that there is a relationship between the theory behind the languages of PDDL, TAL, NDDL, and SAS+, and we plan to identify and describe the relationship so that we can better understand how EUROPA$_2$ compares to these systems.

We plan to extend our modeling language in two ways: 1.

provide better modeling support for time-invariant relationships; 2. provide means to describe optimization criteria. Some of the domains, such as the image processing domain require the specification and reasoning about relationships that are immutable with respect to time. We currently provide the means only to specify data but relationships among the data are assumed to change with respect to time. Furthermore, many planning applications require not only finding a plan but finding a plan with respect to certain optimization criteria. We plan to extend NDDL to allow describing optimization criteria such as minimize makespan or minimize resource consumption.

Finally, we have numerous plans for extending our implementation. We plan to extend the set of planning services provided to include domain analysis techniques such as reachability. We are already working on a PDDL front-end for EUROPA$_2$. Furthermore, we plan to extend the set of services provided by adding direct support for lifted local search planning; more specialized constraint reasoners; and hybrid solvers. The current EUROPA$_2$ implementation has been designed to deal with consistent as well as inconsistent states but only a backtracking planner has been implemented to date. We need to extend the notion of flaws to include violations to be able to handle local search methods, and test whether the implementation holds.

Related Work

EUROPA$_2$ is certainly not the only planner that can plan with resources and express resources as first class citizens. IxTeT already plans with resources, however, IxTeT requires modeling state changing properties as attributes. EUROPA$_2$ allows the expression and reasoning of arbitrary objects, not just objects that behave like attributes. IxTeT, however, provides more reasoning support for resources than CAPR in that they define how to prune "dominated" transaction ordering decisions: 1) they use some graph theory to infer that only certain decisions are necessary, then 2) eliminate "dominated" decision (e.g. if $a < b \rightarrow c < b$ then $a < b$ is not considered.) However, we were unable to find soundness and completeness proofs of planning with resources in IxTeT.

ZENO (Pemberthy & Weld 1994) is a sound and complete planner that handles actions with temporal quantified preconditions and effects. ZENO can reason about deadline goals, piecewise-linear continuous change, external events and, to a limited extent, simultaneous actions. In particular, actions are allowed to overlap in time only when their effects do not interfere. From what we can tell, there is no special purpose reasoning on constraints, and instead, variable assignments ensure that non-linear equations reduce to linear equations. In contrast, EUROPA$_2$ provides 1. a language for expressing declarative resources, 2. ability to express richer types of resources, and 3. ability to handle any type of constraint.

Given the success of PDDL (Fox & Long 2003) in the academic community, PDDL has been extended to cope with problems of increasing size and complexity. However, the extensions have been driven by the capability of planners that have participated in the competitions. PDDL thus, is able to describe plan metrics, a capability that we plan to include in EUROPA$_2$. PDDL, however, has a process-driven time semantics and is unable to deal with preconditions that hold over specific intervals of time and effects that can happen at arbitrary points during action execution. In EUROPA$_2$ resources are first-class citizens and can be fully described declaratively, and relationships between other entities in the plan and resources can be expressed as constraints. In PDDL, resources are reperesented by numeric fluents, which provide advantages and disadvantages. The ability to represent numeric fluents means that planners can then subgoal based on internal numeric states. However, it is difficult and awkward to express a unified view of resources and their properties, which means that planners cannot take advantage of dedicated reasoning algorithms to solve problems with resources. PDDL is a stronger language for specifying goals, e.g. it is possible in PDDL to express goals with disjunctions. In EUROPA$_2$ it is only possible to describe goals in terms of whether they are required or optional, but arbitrary formulae are not allowed.

The Coupled Layered Architecture for Robotic Autonomy CLARATy, is an architecture with goals to: 1. Capture and integrate a wide range of technologies; 2. Leverage existing tools; 3. Leverage experience and tools of the larger software development community; 4. Apply appropriate design patterns to the domain; 5. Provide an infrastructure that enables rapid robotic development; and 6. Capture experience of technologists implementations. The goals of EUROPA$_2$ are very similar. Having deployed the previous generation of EUROPA$_2$ in field tests and missions we have learned that different missions require different functionality, yet they require high performance. EUROPA$_2$ is being developed in order to support the development of generic algorithms, reduce the need for recurring problems for every deployment, simplify the integration of new technologies, use the same framework across deployments, increase functionality by leveraging a more mature base. These are the same motivations that drive CLARATy. CLARATY is a two-layered architecture the first layer is the decision layer that includes the planner, models, and heuristics. The second layer provides the abstraction of the specific robot components. The first layer is based on ASPEN/CASPER system architecture which is similar to EUROPA$_2$'s architecture in that the search engine performs operations on an activity database which in turn performs constraint propagation over parameters and temporal constraints. ASPEN, however, allows you to solve problems using local repair algorithms only. We provide a framework where you should able to implement a local repair planner and a chronological backtracking planner using some the same components.

Acknowledgements

We wish to thank the rest of the EUROPA$_2$ development team: Andrew Bachmann, Will Edgington, Michael Iatauro, and Sailesh Ramakrishnan, for their important contributions to this work. This research was supported by NASA Ames Research Center and the NASA Intelligent Systems program.

References

Bresina, J.; Jónsson, A.; Morris, P.; and Rajan, K. 2003. Constraint maintenance with preferences and underlying flexible solution. In *Constraint Programming Workshop on Change and Uncertainty.*

Despouys, O., and Ingrand, F. 1999. Propice-plan: Towards a unified framework for planning and execution. In *Proceedings of the 5th European Conference on Planning.*

Dias, M.; Lemai, S.; and Muscettola, N. 2003. A real-time rover executive based on model-based reactive planning. In *Proceedings of the International Conference on Robotics and Automation.*

Dvorak, D.; Rasmussen, R.; Reeves, G.; and Sacks, A. 2000. Software architecture themes in "jpl"'s mission data system. In *IEEE Aerospace Conference.*

Fox, M., and Long, D. 2003. Pddl 2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20.

Frank, J., and Jónsson, A. 2003. Constraint based attribute and interval planning. *Journal of Constraints* 8(4).

Frank, J. 2004. Bounding the resource availability of partially ordered events with constant resource impact. In *Proceedings of the* 10^{th} *International Conference on the Principles and Practices of Constraint Programming.*

Golden, K.; Pang, W.; Nemani, R.; and Votava, P. 2003. Automating the processing of earth observation data. In *Proceedings of the* 7^{th} *International Symposium on Artificial Intelligence, Robotics and Space.*

ILOG. 1996. Ilog solver: User manual. Version 3.2.

Jónsson, A.; Morris, P.; Muscettola, N.; Rajan, K.; and Smith, B. 2000. Planning in interplanetary space: Theory and practice. In *Proceedings of the* 5^{th} *International Conference on Artificial Intelligence Planning and Scheduling.*

Laburthe, F., and the OCRE Research Group. 2001. Choco, a constraint programming kernel for solving combinatorial optimization problems. Available at http://www.choco-constraints.net.

Muscettola, N.; Nayak, P.; Pell, B.; ; and Williams, B. 1998. Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence* 103(1-2).

Muscettola, N. 2002. Computing the envelope for stepwise constant resource allocations. In *Proceedings of the* 8^{th} *International Conference on the Principles and Practices of Constraint Programming.*

Nesnas, I.; Wright, A.; Bajracharya, M.; Simmons, R.; Estlin, T.; and Kim, W. S. 2003. Claraty: An architecture for reusable robotic software. In *Proceedings of the SPIE Aerosense Conference.*

Pemberthy, J., and Weld, D. 1994. Temporal planning with continuous change. In *Proceedings of the 12th National Conference on Artificial Intelligence,* 1010–1015.

Tran, D.; Chien, S.; Sherwood, R.; no, R. C.; Cichy, B.; Davies, A.; and Rabbideau, G. 2004. The autonomous sciencecraft experiment onboard the eo-1 spacecraft. In *Proceedings of the* 19^{th} *National Conference on Artificial Intelligence.*

Whalley, M.; Takahashi, M.; Schulein, G.; Freed, M.; Christian, D.; Patterson-Hine, A.; and Harris, R. 2003. The nasa army autonomous rotorcraft project. In *Proceedings of the American Helicopter Society* 59^{th} *Annual Forum,* 61–677.