EUROPA 2: User and Contributor Guide

Tania Bedrax-Weiss Conor McGann Andrew Bachmann
Will Edgington
Michael Iatauro
QSS Group Inc.
Computational Sciences Division
NASA Ames Research Center
Moffett Field, CA 94034-1000
{tania,mcgann,bachmann,wedgingt,miatauro}@email.arc.nasa.gov

January 5, 2005

Contents

1 Introduction

EUROPA 2 is the next generation of the Extensible Universal Remote Operations Architecture. Like the CLARATy robotics control architecture [?], MDS [?], or ILOG [?], EUROPA 2 is a component-based software library for representation and reasoning with plans and schedules. Our goal in developing EUROPA 2 is to provide a fast, flexible, extensible, reusable technology platform for building planning and scheduling applications suitable for space exploration.

EUROPA [6, 2] which has been the core planning technology for a variety of NASA-relevant research and mission applications. A notable example is MAPGEN [?], the ground-based daily activity planning system for the Mars Exploration Rover mission. EUROPA in turn is derived from HSTS which was the planner for the Remote Agent[3]. EUROPA 2 builds on the legacies of EUROPA and HSTS and provides improvements in performance, expressivity, reasoning capabilities, flexibility, extensibility, and modularity which has opened the technology to a broader range of planning techniques (e.g. POCL planning).

1.1 Plan Services

EUROPA 2 provides the following services:

- Domain modeling: for describing planning domains
- Plan representation: for updating partial plans
- Constraint propagation: for propagating the consequences of updates to plans and determining violations
- Subgoaling: for generating consequences of commitments in the plan
- Flaw definition: for specifying flaws from a partial plan
- Decision management: for generating and resolving flaws
- Plan assessment: for determining plan completeness

EUROPA 2 provides the New Domain Description Language NDDL, which deviates substantially from DDL in that it provides an object oriented syntax. It is also compiled, instead of interpreted. NDDL provides syntax for describing objects, timelines, resources, predicates, rules, variables, and constraints. It also provides facilities like inheritance and containment to describe complex objects. Please see Section 4 for more information on NDDL.

Plan representation is a service provided by the Plan Database, much as it was in EUROPA. The Plan Database responds to plan modification operations by updating the partial plan or invoking specialized components to update parts of the partial plan. Constraint propagation, for instance, is a service provided by the Constraint Engine and is in charge of responding to updates to constraints and variables triggered by the plan modification operations. The Rules Engine is in charge of subgoaling also in response to plan modification operations.

Flaw definition, decision management, and plan assesment services are provided by the CBPlanner module. The CBPlanner module implements a chronological backtracking planner that resolves all flaws in the partial plan except for temporal variable assignments as described in [5, 1].

The philosophy underlying EUROPA 2 is to acknowledge up front that no one size fits all when it comes to which techniques to use, and which capabilities to employ. Consequently, EUROPA 2 is engineered to allow people to take just what they need, discard what they do not, and integrate extensions to suit their particular requirements in a straighforward manner. The design strategy is to focus on a core framework defininig the principal abstractions and interactions induced by our underlying paradigm. We then provide concrete components to allow particular assemblies to be defined.

1.2 EUROPA 2 Design Goals

To meet the needs of missions and research projects, the design of EUROPA 2 must be: 1. Efficient to ensure low latency for operations and queries; 2. Flexibile to ensure services can be selected and flexibly integrated; 3. Extensible to ensure services can be enhanced to meet the needs of research or mission applications.

1.2.1 Efficiency

EUROPA 2 has produced significant gains in speed over EUROPA. The primary contributors to the improvement arise from: 1. Fast interfaces and specialized implementations: the ability to tune implementations using inheritance provides speed improvements in key areas such as operations on domains. 2. Efficient merging: EUROPA 2 provides an algorithm to handle merging operations that disables redundant constraints arising in the plan database. 3. Incremental relaxation: when relaxing a variable, EUROPA 2 relaxes only variables reachable through the constraint graph. 4. Direct support for static facts: EUROPA 2 uses objects to capture static facts. Objects can be referenced through variables. We provide a pattern for existentially quantifying objects. By contrast, EUROPA used timelines with a single predicate to capture this information, incurring a high overhead through inefficient merging.

1.2.2 Flexibility

EUROPA 2 is highly customizable. Support for resources may be ommitted if a problem does not require resources. 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. An open API ensures flexbility in how EUROPA 2 is integrated.

1.2.3 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.

The content of this guide is laid out as follows. We begin with an explanation of the concepts underlying EUROPA 2 addressing its role as an embedded technology within a planner, and its underlying paradigms for representation and reasoning. We then switch gears to get the reader up and running with a particular assembly of EUROPA 2 that is included with the distribution. In this section, the reader will solve a prepared planning problem with EUROPA 2, without really understanding much about how it happened! Following this, we seek to build up some understanding with a tutorial-like exposition of model development, and problem solving with EUROPA 2's primary modeling language - NDDL. Having spent some effort working on the application of EUROPA 2, we turn to its underlying architecture. This section gets 'under-the-hood' to provide an understanding of how EUROPA 2 works at the implementation level. Finally, we address the technical aspects of customization and extension. Detailed reference material is included in the appendices.

2 Concepts

2.1 Plans in Planning, Scheduling and Execution

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

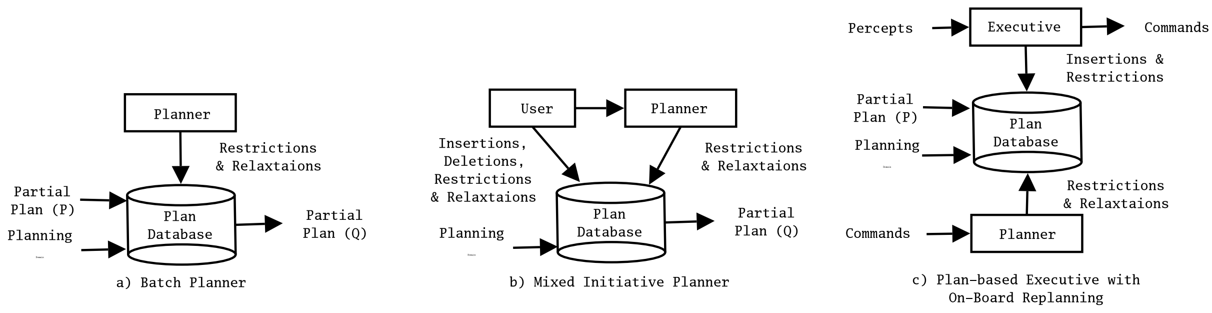
Planner
Restrictions & Relaxtaions
Partial Plan (P)
Planning
...
Plan Database
Partial Plan (Q)
a) Batch Planner
User
Planner
Insertions, Deletions, Restrictions & Relaxtaions
Restrictions & Relaxtaions
Planning
...
Plan Database
Partial Plan (Q)
b) Mixed Initiative Planner
Percepts
Executive
Commands
Insertions & Restrictions
Partial Plan (P)
Planning
...
Plan Database
Restrictions & Relaxtaions
Commands
Planner
c) Plan-based Executive with On-Board Replanning

Figure 1: Sample Plan Database Applications

the cases described, *clients* (i.e. Planner, User, Executive) leverage the services of a common *server*, the *Plan Database*.

2.2 Model-Based Planning

A *Model* expresses laws that govern a particular domain of interest. A model for EUROPA 2 usually contains descriptions of entities in the system and relationships between them, that allow a planner/scheduler to infer conditions that must be satisfied in order to arrive to a solution to a problem based on that model. Domains and problem instances are described in EUROPA 2 in NDDL. A typical NDDL description will contain a set of classes, predicates, and configuration rules. Classes represent properties of the world that may or may not evolve over time; predicates (with variables) represent state descriptions; and configuration rules represent relationships between state descriptions that must hold (the laws of the domain).

2.3 Partial Plans: States and Relationships

A *Partial Plan* represents a set of networks of states typically ordered by time, though it can be ordered by precedence constraints or completely unordered (e.g. a bag of states) in planning, scheduling, and execution. Each network is represented as an *Object*, which corresponds to an instance of a model class. Objects that impose mutual exclusions as well as a total order on its set of states is defined as a *Timeline*. Objects that track numeric change are defined as *Resources*. Objects that are just bags of states are called Objects. EUROPA 2 provides means to specialize object implementations.

States in Objects are represented by *Tokens* and can correspond to activities in scheduling, actions and fluents in planning, and commands in execution. Tokens are instances of predicates, and like predicates, they contain variables that can be used to augment state descriptions. Some of these variables can be temporal variables indicating the temporal extent of the token/state. Relationships among tokens are defined by configuration rules. Configuration rules

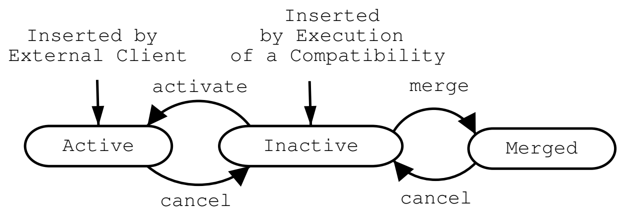
Inserted by External Client
Inserted by Execution of a Compatibility
activate
merge
Active
Inactive
Merged
cancel
cancel

Figure 2: Token State Transition Diagram

specify relationships among tokens (e.g. subgoaling relationships) expressed as constraints between variables of tokens.

2.4 Token State Model

A token in EUROPA 2 evolves as decisions are made. Initially a token may be *active* or *inactive*. A token is *active* if commitment to the token being in the plan is made. A token is *inactive* otherwise. All token flaws that can be inferred from the partial plan and the model via configuration rules are represented as *Inactive Tokens*. Figure 2 illustrates the states and transitions of tokens in EUROPA 2. A token is *Active* immediately when introduced by an actor external to the plan database, as is the case with a goal G specified in an initial partial plan. A token is initially *Inactive* when introduced by a configuration rule of the predicate of an *Active Token*. As prescribed by CAPR [1], an inactive token corresponds to a token flaw which can be resolved by either *merging* with a matching *Active Token* or by choosing to use the resolver $\top$ via *activation*. Merging is chosen to represent that the configuration rule is satisfied by an existing state, that which is active and participates in merging. Activation is chosen to represent that a new state is needed to satisfy the configuration rule.

2.5 Planning/Scheduling Decisions

Planning/scheduling decisions arise from flaws in the partial plan. Flaws can be either unsatisfied configuration rules, unordered tokens in objects (if they have specific ordering requirements), or unbound variable assignments. A planner is done when it has verified that all applicable configuration rules are satisfied. A configuration rule is applicable when there is an active token with a predicate that matches a configuration rule in the model. Unbound variables may occur in tokens, objects, or rules. These must be bound to singletons before the planner can deem the plan complete. There are a few caveats, however. EUROPA 2 provides the ability to plan within a specific horizon. Thus, tokens and variables of tokens that lie outside of the horizon will not be deemed as flaws. By default, the horizon is [-INFINITY, +INFINITY], but the horizon may be specified in the initial state. Another caveat is that we provide a mechanism for specifying precisely the set of flaws that should be resolved before the planner finishes "completing" the partial plan. This mechanism is explained in full in a paper

[5], however, we mention that flaws can be included in the set that need to be resolved via the specification of conditions that flaws must satisfy. Thus, a plan is not complete until all flaws that satisfy all conditions are resolved.

2.6 Early vs. Delayed Commitment

EUROPA 2 provides at ways of controlling early vs. delayed commitment through controlling variable assignments. One example is how EUROPA 2 handles temporal assignments. EUROPA 2 allows representation and reasoning over temporal intervals. This flexibility turns out to be very useful when there's lack of knowledge on precisely when states will hold. Temporal variable assignments can be delayed indefinitely by excluding them from the set of flaws. This can be done via a mechanism we call Plan Identification [5]. In brief, EUROPA 2 provides a set of standard conditions that candidate flaws have to pass in order to become flaws that must be resolved for planning to complete. One of these conditions is a temporal variable condition that filters out all temporal variables, thus, temporal variables never appear as variables whose assignments need to be made.

Another way of controlling early vs. delayed commitment can be done by specifying rules conditional on variable assignments. A heuristic can then be used to determine when such variable assignments should be made, and when subgoaling can occur.

3 Hello Rover - Getting started with PLASMA

This section will demonstrate a simple example that takes an initial state and a model and runs a planner on them to produce a plan. EUROPA 2 does not include a GUI. However, you can use an adjunct project called PlanWorks to visualize a plan and step through the planning process once the planner has produced a plan. This section will illustrate how to get started with EUROPA 2.

3.1 Creating a Project

After successfully building EUROPA 2 by following the instructions in the README and BUILDING files located in the PLASMA root directory, you can create your own project by invoking "makeproject" with the name of your project as an argument in the PLASMA root directory. This will create a directory with the name of your project that is parallel to the PLASMA directory. In it you'll find the following files: Jamrules, Jamfile, ¡Project¿-Main.cc, ¡Project¿-initial-state.nddl ¡Project¿-model.nddl and ppw-config. EUROPA 2 uses Jam instead of make to build its files. Jamrules and Jamfile are both part of the build system. Jamrules specifies some variables and establishes dependencies with EUROPA 2. Jamfile specifies the main program and its dependencies. ppw-config is a file that contains configuration options for PlanWorks. PlanWorks is a plan visualization system that can be installed along with EUROPA 2 to aid in understanding and debugging. ¡Project¿-initial-state.nddl contains the initial state or problem description and ¡Project¿-model.nddl contains the model or domain description. Both files contain descriptions in the NDDL language. Finally, the ¡Project¿-main.cc contains the main program that creates an assembly (configuration of EUROPA 2 components) and plans based on the given initial state and model files.

3.2 Building a simple model

In this section we'll go through the model file and explain it in detail. Your model file should look like this:

```
#include "../PLASMA/NDDL/core/Plasma.nddl"
#include "../PLASMA/NDDL/core/PlannerConfig.nddl"

/**
 * @brief Place holder class with a single predicate
 */
class YourObject extends Timeline {
 predicate helloWorld{} /*!< Predicate with no arguments */
}

/**
 * @brief A simple rule to force a repeated cycle
 */
YourObject::helloWorld{
 eq(duration, 10);
 meets (object.helloWorld);
 met_by(object.helloWorld);
}
```

All models will include Plasma.nddl and PlannerConfig.nddl. Plasma.nddl contains definitions for most common NDDL constructs. PlannerConfig.nddl contains definitions for horizons and the maximum number of steps a planner can take before finding a plan or giving up.

The next section of the model defines the class "YourObject". Notice that we specify "YourObject" as a timeline. Other options are object or resource. YourObject contains a single predicate called "helloWorld" with no arguments. "helloWorld", however, has hidden variables to denote its duration, the class it belongs to, and a few other things. "helloWorld" will be the only predicate that will describe the state of "YourObject".

Finally, the model contains a single rule for the predicate "helloWorld". It specifies that the predicate has a duration 10 and that it "meets" another predicate of the same time and that it is "met_by" another predicate of the same type. Meets and met_by are taken from the Allen Relations [?] and they are inverse of each other and mean that one predicate must be immediately followed by another and vice-versa. You can see all temporal relations that EUROPA 2 supports in Appendix 8.2.

3.3 Creating an initial state

In this section we'll go through the initial state file and explain it in detail. Your initial state file should look like this:

```
#include "<Project>-model.nddl"

// Create a planner configuration instance in PLASMA.
// Horizon Start, Horizon End, MaxPlannerSteps
PlannerConfig plannerConfiguration = new PlannerConfig(0, 1000, 500);

// Sample object
YourObject object = new YourObject();

// Close the the PLASMA Database - no more objects can be created.
close();

// Now place your goals here.
goal(YourObject.helloWorld initialToken);
initialToken.start.specify(0); // Starts at beginning of the horizon

// The planner should take it from here!
```

Your initial state will always include the model it refers to. It is possible to break up a model into several files and include them all at this point. Alternatively, you can include only those parts of the model that are relevant for this initial state.

Next, you need to create an instance of a PlannerConfig object and give it a start time and end time of the planning horizon and the number of steps to use as a bound during planning.

Next, you should create object instances of your classes. In this case, "object" is an instance of "YourObject". It is possible to attach static properties to objects in the form of variables. If you want to create different object instances with different properties, you may specify a constructor for the object that takes in the different properties as argument.

Once you have finished creating all objects in your system, you must close the database. Since EUROPA 2 is at its core a dynamic constraint satisfaction system, it needs to know the complete set of entities before it can reason with them. Reasoning is suspended while the database is not closed. Once closed there is no way to "open" it.

Finally, you must specify the tokens that you know must be present in the plan. In this example we specify a single token "initialToken" of type "helloWorld" that must be present on some object (in this case there's only one “object”). We also specify that it must start at time 0 via the “eq” constraint which is identified as an equality constraint in the assembly.

3.4 Running the planner

Now that you understand the model and the initial state, can you guess what the plan should look like? Let’s see, the planner should plan for a horizon betwen 0 and 1000. The initial state specifies that there’s a single object “object” with a single token “initialState” of type “helloWorld” that starts at time 0. Since “helloWorld” has duration 10, which we know from the model, and since “helloWorld” must meet and be met_by another “helloWorld” token, we can begin to hypothesize that the end result will be an “object” full of helloWorlds, all abutting each other. How many? Well, we should see 100 of them, since each has duration 10.

To see this in action, let’s run by invoking “jam ¡Project¿” in the ¡Project¿ root directory. If you haven’t already built EUROPA 2, this should trigger a build. This command will also trigger a build of your main program. You’ll see a target called ¡Project¿_g_rt, and a file with the output plan called RUN_¡Project¿-planner_g_rt.¡Project¿-initial-state.xml.output. Your output file should show the “object” with a sequence of “helloWorld” tokens lying between 0 and 1000. Your output file should look like this (we’ve replaced some of the output by ...):

```
Found a plan at depth 299 after 299 nodes.
PlannerConfig:plannerConfiguration*************************
YourObject:object*************************
[ INT_INTERVAL:CLOSED[0, 0] ]
YourObject.helloWorld()
Key=22
Merged Key=101
[ INT_INTERVAL:CLOSED[10, 10] ]
[ INT_INTERVAL:CLOSED[10, 10] ]
YourObject.helloWorld()
Key=41
Merged Key=148
[ INT_INTERVAL:CLOSED[20, 20] ]
[ INT_INTERVAL:CLOSED[20, 20] ]
YourObject.helloWorld()
Key=85
Merged Key=200
...
[ INT_INTERVAL:CLOSED[970, 970] ]
YourObject.helloWorld()
Key=5020
Merged Key=5140
[ INT_INTERVAL:CLOSED[980, 980] ]
[ INT_INTERVAL:CLOSED[980, 980] ]
```

```
YourObject.helloWorld()
Key=5072
Merged Key=5192
[ INT_INTERVAL:CLOSED[990, 990] ]
[ INT_INTERVAL:CLOSED[990, 990] ]
YourObject.helloWorld()
Key=5124
[ INT_INTERVAL:CLOSED[1000, 1000] ]
End Timeline: object*************************
Inactive Tokens: *************************
[ INT_INTERVAL:CLOSED[-inf, -1] ]
YourObject.helloWorld()
Key=57
[ INT_INTERVAL:CLOSED[0, 0] ]
[ INT_INTERVAL:CLOSED[1000, 1000] ]
YourObject.helloWorld()
Key=5176
[ INT_INTERVAL:CLOSED[1001, +inf] ]
Merged Tokens: *************************
[ INT_INTERVAL:CLOSED[-inf, 9] ]
YourObject.helloWorld()
Key=101
[ INT_INTERVAL:CLOSED[10, 10] ]
[ INT_INTERVAL:CLOSED[-inf, 19] ]
YourObject.helloWorld()
Key=148
[ INT_INTERVAL:CLOSED[20, 20] ]
...
[ INT_INTERVAL:CLOSED[-inf, 979] ]
YourObject.helloWorld()
Key=5140
[ INT_INTERVAL:CLOSED[980, 980] ]
[ INT_INTERVAL:CLOSED[-inf, 989] ]
YourObject.helloWorld()
Key=5192
[ INT_INTERVAL:CLOSED[990, 990] ]
Finished
```

You'll notice that the file shows the resulting state of the objects declared in the initial state. You'll notice also that there are two inactive tokens, one that falls before the first token in "object" and another that falls after the last token in "object". These are tokens that fall outside of the horizon and are therefore, not flaws of the plan, which is why they remain inactive. Notice also that there's a list of merged tokens. Some of these tokens have unbound variables. These are variables that were not decided before the planner decided to merge the token. Once a token is merged, its variables are no longer affected by propagation, for efficiency reasons.

3.5 Visualization of the plan and planning process in PlanWorks

We will not go into all the details. Refer to PlanWorks documentation for setup instructions of use. We may wish to include screen shots.

3.6 Debugging

We should mention Debug.cfg and explain how it works.

4 Developing Your Own Model In NDDL

In this section you will learn how to develop your own model in NDDL. We will show most of the NDDL constructs but please refer to Appendix 8.1 for the complete list.

4.1 Rover: The Robotic Geologist

We use a planning domain loosely based on the MER mission to show the services provided by EUROPA 2. We assume the application in question is one of producing daily activity plans for operation of a robotic planetary surface geologist we call *Rover*. *Rover* is a mobile robot equiped with a range of instruments to sample and study a geological site. We will focus on the panoramic imager *PanCam*. A *Rover* has a battery on board, and can replenish its energy levels using solar power. A *Rover* will be given activities to travel to a specific location, deploy an instrument at that location, and perform an experiment. It will also be given information from the terrain. For simplicity purposes we assume that the terrain is represented by a euclidean grid and that locations are indentified with two coordinates. We also assume that paths between locations are fixed and that the Rover can only move between locations if there is a path between them. Furthermore, the Rover can only access those locations that it can reach without draining the battery.

The Rover includes a number of components that can be operating concurrently. For example, the *PanCam* can be tracking targets while the Rover is driving. The Rover also imposes mutual exclusion constraints on components, for instace, the rock abrasion tool *RAT* must be stowed while Rover is moving. Furthermore, given a command to deploy a particular instrument at a specific location, the Rover needs to insure the following occurs in order: The instrument must be first unstowed, and then positioned. Then, if it decides not to perform another experiment with the same instrument, it must stow the instrument.

The model must be carefully crafted so that all component interactions are modeled so that correct command sequences can be inferred from this model.

In the next few sections we will show how to model each of the following model components: locations and paths, instruments, batteries, and rovers. Then, we'll show you how to connect all of these components and describe rules that govern the components and interactions. Finally, we'll create a small initial state that you'll be able to run with your new model.

4.2 Locations and Paths

When we model locations we're faced with a choice of modeling locations as an enumerated set and not representing the coordinates, or to model locations as objects with coordinates as properties that are static with respect to time. We will show both ways of modeling locations.

If we decided to choose to ignore properties of locations and instead enumerate all possible locations we would include the following in the model:

```
enum Location { loc1, loc2, loc3 };
```

EUROPA 2 would interpret this as a new user-defined type with values "loc1", "loc2" and "loc3".

Alternatively, we can use a class to define locations and annotate locations with their properties. Furthermore, we don't have to express in the model how many locations we'll have, it can be specified in a problem instance. In this case, we would write the following:

```
class Location {
  int x;
  int y;
  string label;
  Location(int _x, int _y){
    x = _x;
    y = _y;
    label = ``anonymous'';
  }
  Location(int _x, int _y, string _label){
    x = _x;
    y = _y;
    label = _label;
  }
}
```

Notice that a location has three static properties: x coordinate, y coordinate, and a label. Notice there are two constructors, one that takes in a label on construction and the other that assigns "anonymous" to the label. It is often useful to assign arbitrary data to domain elements. Domain elements that are described with static data only are called static and don't evolve over time. Locations are examples of static data.

Paths are also static domain elements. Let's assume that paths have a name to identify them, a cost, and the two locations it links. We can describe paths in the model in the following way:

```
class Path {
  string name;
  Location from;
  Location to;
  float cost;
  Path(string _name, Location _from, Location _to, float _cost) {
    name = _name;
    from = _from;
    to = _to;
    cost = _cost;
  }
}
```

Notice that this class is composed not only of primitive types but also of user-defined types (i.e., Location). So far, we have introduced the following primitive types: int, float, string. We have also introduced enumerations. Any class can contain a member of an enumerated type. Say that paths were classified into difficulty levels depending on the obstructions along the way. Let's say that we have a total of 5 difficulty levels: low, low-medium, medium, medium-high, and high. We would modify the description above to include the category in the following way:

```
enum Category { low, low-medium, medium, medium-high, high }
class Path {
  string name;
  Location from;
  Location to;
  float cost;
  Category level;
  Path(string _name, Location _from, Location _to, float _cost, Category _level ) {
    name = _name;
    from = _from;
    to = _to;
    cost = _cost;
    Category level = _level;
  }
}
```

4.3 Instruments

Instruments have state that can evolve over time. There are two classes of instruments. Instruments that can be stowed for protection from the elements and instruments that are permanently exposed. This example shows how we can aggregate the common properties of all instruments into a class and then use inheritance to derive the more specific class of stowable instruments. All instruments can be positioned, but only stowable instruments can be stowed and unstowed. Furthermore, we will assume that the speed that an instrument can be stowed and unstowed with varies depending on the kind of instrument. The speed of placement will depend on the location. In the model, we would write the following:

```
class Instrument {
  predicate Position {
    Location rock;
    int position_speed;
    eq(position_speed, duration);
  }
  predicate Positioned { }
}

class Stowable_Instrument extends Instrument {
  int stow_speed;
  int unstow_speed;
  Stowable_Instrument(int _stow_spd, int _unstow_spd) {
    stow_speed = _stow_spd;
    unstow_speed = _unstow_spd;
  }
  predicate Stow { }
  predicate Unstow { }
  predicate Stowed { }
  predicate Unstowed { }
}
```

Stowable instruments will inherit the predicates "Position" and "Positioned" from "Instrument", since these are the only members of the class. Note that predicates can have argument variables such as "position_speed" from the "Position" predicate. Predicates can also impose constraints by definition on its arguments whether they're implicit, such as duration, or explicit, such as "position_speed". In this case, we equal the corresponding speeds to the duration of each of the "Position", "Stow", or "Unstow" predicates.

4.4 Batteries

The Rover has a main battery that is solar-powered. A battery has state that is represented by numerical quantities and as such is represented in NDDL as a resource type. In NDDL one can represent resources with different properties. If we assume that the battery cannot be replenished, we say that the battery is a consumable resource [4]. Resources have a default constructor and can be initialized with the following parameters (for full treatment of resources representation see [4]):

1. "initial capacity": the initial capacity of the resource
2. "level limit min": the minimum level that the resource is allowed to reach
3. "level limit max": the maximum level that the resource is allowed to reach
4. "production rate max": the maximum production rate per unit of time
5. "production max": the maximum production overall
6. "consumption rate max": the maximum consumption rate per unit of time
7. "consumption max": the maximum consumption overall

When creating a specific type of resource, one can call the constructor of a resource "super" with specific values for these parameters to set the type of resource. For example, if Battery is a consumable resource we would write the following in the model:

```
class Battery extends Resource {
  Battery(float initial_capacity, float level_limit_min, float level_limit_max) {
    super(initial_capacity, level_limit_min, level_limit_max, 0.0, 0.0,
          MINUS_INFINITY, MINUS_INFINITY);
  }
}
```

Note that we use the special constants "MINUS_INFINITY" and "PLUS_INFINITY" to indicate that there are no restrictions on the amount that is consumed overall or the rate at which it is consumed.

4.5 Rovers

Rovers are composed of two kinds of instruments, a *PanCam* which is an instrument, and a *RAT* which is a stowable instrument. A rover must move between locations. In order to track the rover's position, we chose to model the navigation component separately. The navigation component will be modeled in NDDL as follows:

```
class Navigator {
  predicate At {
    Location location;
  }
  predicate Going{
    Location from;
    Location to;
    neq(from, to);
  }
}
```

The “Navigator” component indicates that the Rover can either be “At” a location or “Going” from one location to another. Furthermore, a Rover can only move between different locations as indicated by the constraint “neq(from, to);”. This constraint can appear in the predicate definition because it references only variables that are members of the predicate. If you want to include constraints between things other than predicate arguments (whether explicit or implicit such as duration or object) you must do so in a rule. We’ll show how when we describe rules in Section 4.6.

Finally, we can describe a Rover in the following way:

```
class Rover {
  Navigator nav;
  Instrument pancam;
  Stowable_Instrument rat;
  Battery battery;
  Rover(Battery r) {
    nav = new Navigator();
    pancam = new Instrument();
    rat = new Stowable_Instrument();
    battery = r;
  }
}
```

4.6 Rules

We are now in a position to write the rules governing the state transitions for Instruments and Navigators. Let’s first go through the rules:

1. In order to position an instrument, the navigator has to be at the destination.
2. Placing an instrument consumes 20 units of the battery.
3. In order to unstow an instrument, the navigator cannot be moving, thus, it has to be At some location.
4. Unstowing is possible only if the instrument was stowed and after unstowing the instrument will be unstowed.
5. Unstowing consumes 20 units of battery.
6. In order to stow an instrument, the navigator cannot be moving, thus, it has to be At some location.
7. Stowing is possible only if the instrument was positioned (we don’t want to allow unstowing and stowing for no reason) and after stowing the instrument will be stowed.

8. Stowing consumes 20 units of battery.
9. Stowed is preceeded by a Stow and followed by an Unstow.
10. Unstowed is preceeded by an Unstow and followed by a Position.
11. At is preceeded and followed by a Going such that the preceeding Going's destination is the source of At and the successor Going token's source is the source of At.
12. Going is preceeded by an At such that the location of at is the from of Going, and Going is also succeeded by an At such that the location of at is the to of the Going.
13. Going between a to and a from is only allowed if there exists a path p between the to and the from.
14. Going consumes as much battery as the cost of the path.

Rules that apply to the same predicate can be aggregated into the same rule. For instance, the rules above in the model would appear as:

4.6.1 Basic Rules

```
// Rules for Instrument

Instrument::Position{
 // make sure it is at the destination during the execution of Position
 contained_by(Navigator.At at);
 eq(at.location, destination);
 // ensure that both at and position refer to the same object.
 Rover rovers;
 commonAncestor(at.object, this.object, rovers);
 // consume battery
 starts(Battery.change tx);
 eq(tx.quantity, -20);
}

// Rules for Stowable_Instrument

Stowable_Instrument::Unstow{
 // make sure it is at some location during the execution of Position
 contained_by(Navigator.At at);
 // ensure that both at and unstow refer to the same object.
 Rover rovers;
 commonAncestor(at.object, this.object, rovers);
 // unstow is followed by unstowed and preceeded by stowed.
 meets(Unstowed a);
 met_by(Stowed b);
 // consume battery
 starts(Battery.change tx);
 eq(tx.quantity, -20);
}

Stowable_Instrument::Stow{
 // make sure it is at some location during the execution of Position
 contained_by(Navigator.At at);
 // ensure that both at and stow refer to the same object.
 Rover rovers;
 commonAncestor(at.object, this.object, rovers);
 // unstow is followed by stowed and preceeded by position.
 meets(Stowed a);
 met_by(Position b);
 // consume battery
 starts(Battery.change tx);
 eq(tx.quantity, -20);
}

Stowable_Instrument::Stowed{
 // stowed is followed by unstow and preceeded by stow.
 met_by(Stow a);
 meets(Unstow b);
}

Stowable_Instrument::Unstowed{
 // unstowed is followed by position and preceeded by unstow.
 met_by(Unstow a);
 meets(Position b);
}

// Rules for Navigator

Navigator::At{
 // at is preceeded by a Going whose destination is location.
 met_by(object.Going go_before);
 eq(go_before.to, location);
 // at is followed by a Going whose source is location.
 meets(object.Going go_after);
 eq(go_after.from, location);
}

Navigator::Going{
 // going is preceeded by an At whose location is from
 met_by(object.At at_before);
 eq(at_before.location, from);
 // going is followed by an At whose location is to
 meets(object.At at_after);
 eq(at_after.location, to);

 // Select a path from those available between the 2 points
 Path p : {
  eq(p.from, from);
  eq(p.to, to);
 };

 // Consume units of battery (assuming p.cost is negative). Should be based
 // on path length.
 starts(Battery.change tx);
 eq(tx.quantity, p.cost);
}
```

4.7 Advanced Rules

In the the previous section we covered:

1. basic rules - meets, met_by, contained_by, starts
2. constraints among parameters of predicates
3. commonAncestor constraint that is used when traversal of the object hierarchy is required
4. existential quantification by creating a filter

In this section we will cover other more advanced concepts such as conditional rules, rule local variables and universal quantification.

Assume now that we want to model a rover that can accept requests to shoot images and shoot them either with a filter or without any filters. Whether to use a filter or not is determined by the image request. Let us first model a camera. We can take advantage of the Instrument class and extend it with the properties we want cameras to have. In NDDL we would write:

```
class Camera extends Instrument {
  predicate ShootRequest {
    bool filtering;
    Location view;
    int length;
  }
  predicate PlaceFilter {
    Location view;
  }
  predicate Shoot {
    Location view;
  }
}
```

Notice that an image request requires that we specify whether or not to use a filter while shooting. Furthermore, each of the predicates keeps track of the location of the image that is to be taken. We will need the navigator component to be at the location and remain at that location through the shoot. The “Location” parameter of all predicates allows us ensure that we stay at the location through the entire shoot. Here are the corresponding rules:

```
Camera::ShootRequest {
 eq(length,duration);
 contained_by(Navigator.At atc);
 eq(atc.location, view);
 if (filtering == true) {
   meets(object.PlaceFilter s0);
   eq(s0.view,view);
 }
 if (filtering == false) {
   meets(object.Shoot s1);
   eq(s1.view,view);
 }
}

Camera::PlaceFilter {
 eq(duration,5);
 contained_by(Navigator.At atc);
 eq(atc.location, view);
 met_by(object.ShootRequest s0);
 eq(s0.view,view);
 meets(object.Shoot s1);
 eq(s1.view,view);
 starts(Battery.change tx);
 eq(tx.quantity, -20);
}

Camera::Shoot {
 contained_by(Navigator.At atc);
 eq(atc.location, view);
 meets(object.ShootRequest s0);
 eq(s0.view,view);
 starts(Battery.change tx);
 eq(tx.quantity, -100);
}
```

4.8 Recap

The entire model described above can be found under UserGuideRover.nddl in the same directory as this user guide. We have introduced the following concepts and illustrated them with examples in the UserGuideRover model:

1. class
2. enum
3. float, int, string
4. inheritance
5. predicates
6. predicate parameter constraints
7. basic rules - meets, met_by, contained_by, starts
8. constraints among parameters of a predicate.
9. commonAncestor constraint that is used when traversal of the object hierarchy is required
10. composition
11. rules
12. guards
13. Using existential quantification - filtering and binding
14. local variables
15. instantiation (not introduced)
16. assigment (in initial state)
17. Using universal quanitifcation - iteration over objects (not introduced)

4.9 Formulate and solve a planning problem

End the section by referencing a model and initial state that can be run through the planner and visualized in PlanWorks.

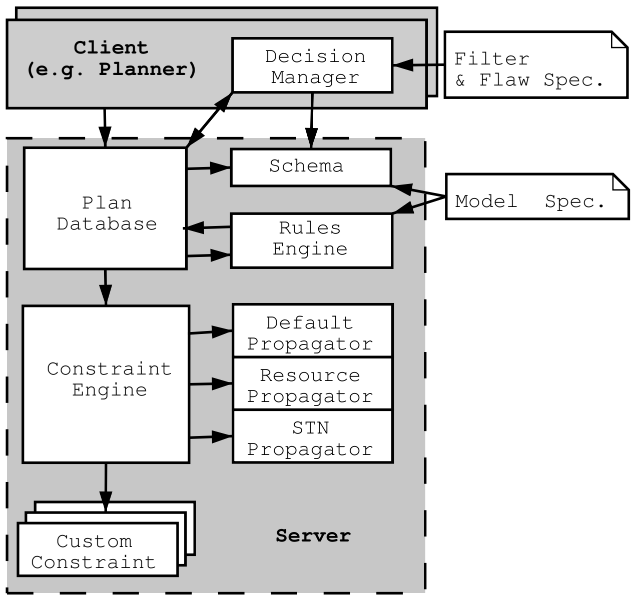

Figure 3: EUROPA 2 System Architecture Diagram

5 PLASMA System Architecture

Figure ?? 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 set of plan services of the server at the abstraction level of primitives in CAPR i.e. tokens, transactions, constraints, resources, variables. The *Constraint Engine* and related components propagate constraints among variables and detect violations. The provided constraints and propagators can be freely integrated or omitted. The *Rules Engine* reacts to 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 view specification to manage the set of flaws for a client.

Picture of overall architecture. Document each component at the level of its main roles and responsibilities. This should be used as an introduction to the API documentation. Ideally, we could directly leverage that documentation.

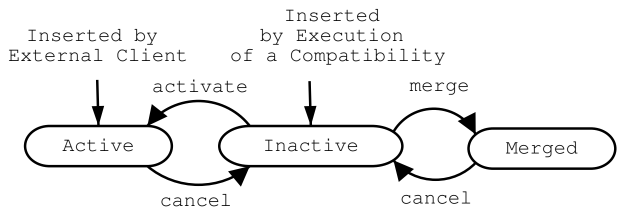

Figure 4: Token State Transition Diagram

5.1 Plan Database

5.2 Constraint Engine

5.3 Temporal Network

5.4 Rules Engine

5.5 Resources

5.6 NDDL Parser and Compiler

5.7 Utilities

5.8 CBPlanner

5.9 Key use cases

Helps to understand the interaction among components. May want to use a simple example model, possibly cut-down from k9.

1. Creating an Object
2. Token activation
3. Token deactivation
4. Constraining a Token
5. Freeing a Token
6. Binding a Variable
7. Freeing a Variable
8. Copying a plan database

6 Customization and Extension

6.1 Configuration and Assembly

6.2 Using and extending the CBPlanner

6.3 Custom constraints

6.4 Custom propagation

6.5 Building model specializations

6.6 Custom rule implementations

6.7 Specialized domains

6.8 External data integration

6.9 Listeners and Loggers

6.10 Integration to PlanWorks

7 Contributing to EUROPA 2

This section should have guidelines for how to contribute to the code base. Where to submit it to, how to contact the primes, how to ask for help, etc. It should also reference the coding guidelines appendix.

References

[1] Andrew Bachmann et. al. EUROPA2: Plan database services for planning and scheduling applications. In *ICAPS*, 2005. Submitted for Publication.

[2] Ari K. Jonsson et. al. Planning in interplanetary space: Theory and practice. In *AIPS*, 2000.

[3] Nicola Muscettola et. al. Remote agent: to boldly go where no AI system has gone before. *Artificial Intelligence*, 1998.

[4] Tania Bedrax-Weiss et. al. Formalizing resources for planning. In *ICAPS Workshop on PDDL*, Italy, 2003.

[5] Tania Bedrax-Weiss et. al. Identifying executable plans. In *ICAPS Workshop on Plan Execution*, Italy, 2003.

[6] Jeremy Frank and Ari K. Jonsson. Constraint based attribute and interval planning. *Journal of Constraints*, 2003.

8 Appendices

8.1 Appendix A: NDDL Language Reference

class predicate int float string enum extends concurrent precedes object start end duration state time new goal rejectable activate specify constrain close

8.2 Appendix B: Temporal Relations

In NDDL, two tokens can be constrained in several simple ways, including with the following “abbreviations” for several commonly used constraints. In the examples, “A” and “B” are tokens. “A.start” is therefore the time at which token A starts; similarly, “B.end” is the time at which token B ends.

NDDL syntax	Constraints created
A after B;	B.end ¡= A.start
A any B;	(none)
A before B;	A.end ¡= B.start
A contained_by B;	B.start ¡= A.start, A.end ¡= B.end
A contains B;	A.start ¡= B.start, B.end ¡= A.end
A contains_end B;	A.start ¡= B.end, B.end ¡= A.end
A contains_start B;	A.start ¡= B.start, B.start ¡= A.end
A ends B;	A.end = B.end
A ends_after B;	B.end ¡= A.end
A ends_after_start B;	B.start ¡= A.end
A ends_before B;	A.end ¡= B.end
A ends_during B;	B.start ¡= A.end, A.end ¡= B.end
A equal B;	A.start = B.start, A.end = B.end
A meets B;	A.end = B.start
A met_by B;	A.start = B.end
A paralleled_by B;	B.start ¡= A.start, B.end ¡= A.end
A parallels B;	A.start ¡= B.start, A.end ¡= B.end
A starts B;	A.start = B.start
A starts_after B;	B.start ¡= A.start
A starts_before B;	A.start ¡= B.start
A starts_before_end B;	A.start ¡= B.end
A starts_during B;	B.start ¡= A.start, A.start ¡= B.end

Here are the inverse relations.

NDDL syntax	Inverse relation
A after B;	A before B;
A any B;	A any B;
A before B;	A after B;
A contained_by B;	A contains B;
A contains B;	A contained_by B;
A contains_end B;	A ends_during B;
A contains_start B;	A starts_during B;
A ends B;	A ends B;
A ends_after B;	A ends_before B;
A ends_after_start B;	A starts_before_end B;
A ends_before B;	A ends_after B;
A ends_during B;	A contains_end B;
A equal B;	A equal B;
A meets B;	A met_by B;
A met_by B;	A meets B;
A paralleled_by B;	A parallels B;
A parallels B;	A paralleled_by B;
A starts B;	A starts B;
A starts_after B;	A starts_before B;
A starts_before B;	A starts_after B;
A starts_before_end B;	A ends_after_start B;
A starts_during B;	A contains_start B;

The names used for the relations as defined in James Allen's original paper [] are:

Allen Relation	Constraints	Equivalent Europa
A equals B (=)	A.start = B.start, A.end = B.end	A equal B
A precedes B (¡)	A.end ¡ B.start	A before [1 +Inf] B
A follows B (¿)	A.start ¿ B.end	A after [1 +Inf] B
A meets B (m)	A.end = B.start	A meets B
A inverse-meets B (im)	A.start = B.end	A met_by B
A during B (d)	A.start ¿ B.start, A.end ¡ B.end	A contained_by [1 +
A inverse-during B (id)	A.start ¡ B.start, A.end ¿ B.end	A contains [1 +Inf]
A starts B (s)	A.start = B.start, A.end ¡ B.end	A parallels [0 0] [-In
A inverse-starts B (is)	A.start = B.start, A.end ¿ B.end	A parallels [0 0] [1 +
A finishes B (f)	A.start ¿ B.start, A.end = B.end	A parallels [-Inf -1]
A inverse-finishes B (if)	A.start ¡ B.start, A.end = B.end	A parallels [1 +Inf]
A overlaps B (o)	A.start ¡ B.start, A.end ¿ B.start, A.end ¡ B.end	¡unsupported as a si
A inverse-overlaps B (io)	A.start ¡ B.end, A.end ¿ B.start, A.end ¿ B.end	¡unsupported as a si

Note the explicit naming of nearly all of the inverse relations. The exceptions are "equals" (its own inverse) and "precedes" and "follows", which are each other's inverse.

Note also the need for explicit bounds in most of the equivalencies due to Europa's relations being based on, e.g., "before or at the same time" rather than Allen's relations being strictly "before" and the lack of explicit support for the last two Allen relations. Europa II's additional flexibility does allow them to be expressed, though more verbosely, as the constraints themselves. E.g.:

Allen Relation	NDDL Constraints
A overlaps B (o)	LessThan(A.start, B.start); LessThan(B.start, A.end); LessThan(A.end, B.end);
A inverse-overlaps B (io)	LessThan(A.start, B.end); LessThan(B.start, A.end); LessThan(B.end, A.end);

8.3 Appendix C: Constraint Library Reference

8.4 Appendix D: Test Language Specification and Use

8.5 Appendix E: Coding Guidelines

General Practices

- Ensure you declare variables and methods in their narrowest scope.
- If you declare a static variable inside a non-static method, double check that the method should not be static and also double check that the variable should not be a member of the class.
- We discourage writing code in header files unless needed for templates or proven performance.
- Use STL classes and methods unless what you need is not provided. Same goes for any other code. Reuse as much as possible.

Pre-processing

- Include system headers by using the angle bracket style. (#include ¡stdio¿)
- Include user files by using the double quote style. (#include "File.h")
- Do not define your own pre-processor macros to control level of or presence of debugging output or error checks.

Namespaces

- Use the std:: prefix, or 'using namespace std;' when using STL.
- Put Europa code in the Europa namespace.

Global Constants

- Use DEFINE_GLOBAL_CONSTANT and DECLARE_GLOBAL_CONSTANT for globals.

Static Class Members

- When handling static data, you must provide an automatic purge mechanism or provide an explicit purge method.

Module Initialization and Termination

- We should standardize method calls to initialization and termination methods. Such as nddl initialization which cascades onto constraint engine initialization.

Iterator Use

- Use const iterators unless you have to use a non-const iterator.
- When using const iterators, use ++iterator rather than iterator++.

Pointer References

- Direct pointer references are discouraged; use class Id instead.
- When creating a reference, create an m_id member that holds the id that gets constructed in the constructor initializer, in the destructor the m_id should be removed.
- When deleting references to ids call delete on the cast operator (e.g. delete (ConstrainedVariable *) ref).

Magic Numbers

- Define an enumerated type to handle number references instead of using magic numbers.

Classes

- Capitalize names of classes. When composing names for classes capitalize the first letter of each word.
- Declare a virtual destructor.

Pure Virtual Classes

- Declare a protected constructor.
- Declare all functions pure virtual.

Methods

- Declare a method const where possible.
- Do not return bare pointers or non-const references.
- If the caller can own a data structure that is to be populated in the callee, create the data structure in the caller and then pass it by reference as an argument.
- Avoid copying of data structures where possible.
- Declare non-primitive arguments as const references.
- Return non-primitive values as const references.

Error Checks

- Use checkError to express pre-conditions.
- Use checkError to express invariants.
- Use checkError to express post-conditions.
- Avoid using non-const functions in checkError tests.
- Do not use assert.
- Do not use Id::isValid outside of checkError.
- Do not write "if (Test) checkError(...Check...);". Write "checkError(!Test —— ...Check...)".

Debugging Output

- Use the Europa debugging output management system.
- Do not put debugging output into stdout or stderr.

Documentation

- Use doxygen style comments with the javadoc style keywords. (@brief, etc.)
- Enforce emacs macros
- Class descriptions, file descriptions, parameters, method, return values, errors
- Documentation is required for header files, recommended for implementation files

9 Acknowledgements

This research was supported by NASA Ames Research Center and the NASA Intelligent Systems program.

References

[1] Andrew Bachmann et. al. EUROPA2: Plan database services for planning and scheduling applications. In *ICAPS*, 2005. Submitted for Publication.

[2] Ari K. Jonsson et. al. Planning in interplanetary space: Theory and practice. In *AIPS*, 2000.

[3] Nicola Muscettola et. al. Remote agent: to boldly go where no AI system has gone before. *Artificial Intelligence*, 1998.

[4] Tania Bedrax-Weiss et. al. Formalizing resources for planning. In *ICAPS Workshop on PDDL*, Italy, 2003.

[5] Tania Bedrax-Weiss et. al. Identifying executable plans. In *ICAPS Workshop on Plan Execution*, Italy, 2003.

[6] Jeremy Frank and Ari K. Jonsson. Constraint based attribute and interval planning. *Journal of Constraints*, 2003.