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,cmcgann,bachmann,wedgingt,miatauro}@email.arc.nasa.gov

February 2, 2005

Contents

1 Introduction

EUROPA 2 is the next generation of the Extensible Universal Remote Operations Architecture. Like the CLARATy robotics control architecture [9], MDS [1], or ILOG [8], 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 [7, 3] 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 [4]. 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

Each of these services can be used independently or in conjunction to build applications by creating an assembly which composes the services needed. Please see Section 6.1 for more information.

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 the literature [6, 2].

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 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 con-

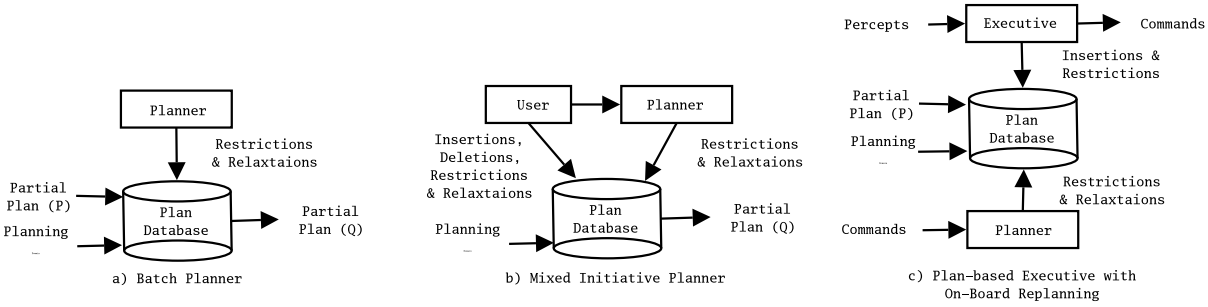

Figure 1: Sample Plan Database Applications

figuration 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 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

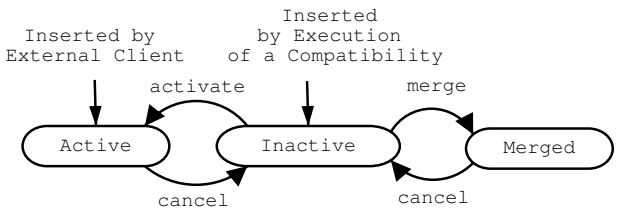

Figure 2: Token State Transition Diagram

is initially *Inactive* when introduced by a configuration rule of the predicate of an *Active Token*. As prescribed by CAPR [2], 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 [6], 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 [6]. 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.

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 [5]. Resources have a default constructor and can be initialized with the following parameters (for full treatment of resources representation see [5]):

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 Basic 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:

```
// 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
 // we create a local variable p to contain all paths that match the from
 // and to.
 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. local rule variables
5. existential quantification by creating a filter

In this section we will cover other more advanced concepts such as conditional rules 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);
}
```

Finally, we illustrate the use of universal quantification by showing how it is possible to quantify over all objects (assuming there's a fixed number of it that is known at start time). Assume the navigator has numerous instruments and you want to impose a constraint that says that all instruments must be in state StandBy whenever the navigator is in state Going. You can either enumerate all instruments and impose the constraint on each one, or you can say so with a shorthand rule as follows:

```
Navigator::Going {
 met_by(object.At at_before);
 eq(at_before.location, from);
 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);
 };
 // Pull juice from the battery. Should be based on path length.
 starts(Battery.change tx);
 eq(tx.quantity, p.cost);

 // Ensure all instruments are in standby
```

```
Rover myRover;
eq(myRover.nav,object);
if (myRover) { // ensure that the object has been bound to a singleton
  Instrument instruments;
  commonAncestor(instruments,this.object,myRover); // instrument on rover
  foreach (i in instruments) {
    contained_by(Instrument.StandBy s); // impose constraint between Going
// and Standby
    eq(s.object,i);
  };
}
```

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. local variables
14. Using existential quantification - filtering and binding
15. Using universal quanitifcation - iteration over objects

4.9 Formulate and solve a planning problem

Let us assume we're given a problem with a lander and four rocks with their corresponding coordinates. Furthermore, let's assume we're given three paths, a very long path, a moderately long path and a short cut. Let's assume that there is a single rover called spirit that carries a battery and the default instruments.

We're interested in finding a plan where spirit is initially at the lander and it has to take an image of rock4 and place an instrument at rock4. Furthermore, assume that our planning horizon is 0 to 1000 and that we're hoping to find a plan within 500 steps.

The initial state would look something like the following:

```
#include "UserGuideRover.nddl"

PlannerConfig plannerConfiguration = new PlannerConfig(0,1000,500);

Location lander = new Location(0, 0, "lander");
Location rock1 = new Location(9, 9, "rock1");
Location rock2 = new Location(1, 6, "rock2");
Location rock3 = new Location(4, 8, "rock3");
Location rock4 = new Location(3, 9, "rock4");

// Allocate paths
Path p1 = new Path("Very Long Way", lander, rock4, -2000.0, Category.'low');
Path p2 = new Path("Moderately Long Way", lander, rock4, -1500.0, Category.'medium');
Path p3 = new Path("Short Cut", lander, rock4, -400.0, Category.'high');

// Allocate Rover
Battery battery = new Battery(1000.0, 0.0, 1000.0);
Rover spirit = new Rover(battery);

close();

// Establish the initial position for spirit
goal(Navigator.At initialPosition);
initialPosition.start.specify(0); // Starts at the beginning of the horizon
initialPosition.location.specify(lander); // Initial position is lander

// Establish a goal - to shoot an image at a location sometime before the
// experiment that follows
goal(Camera.ShootRequest image);
image.filtering.specify(true);
image.view.specify(rock4);
image.length.specify(10);
leq(image.start,49);
leq(0,image.start);
```

```
// Establish a goal - to position an instrument at a location
goal(Stowable_Instrument.Position experiment);
experiment.start.specify(50);
experiment.destination.specify(rock4); // Want to get to rock4
```

The plan we're looking for will contain the steps required to get the stowable instrument from the stand by state to the positioned state. Furthermore, spirit's navigator will move from the lander to rock4, and its camera will take a picture of rock4 with a filter.

4.10 Visualization of the plan and planning process in PlanWorks

EUROPA 2 includes logging facility to transfer data about the planning process to a visualization tool called PlanWorks. PlanWorks reads the data into a set of mysql tables and then produces a visualization of each of the planning steps. PlanWorks can be configured to load all steps at once or only a subset of the steps. A subset is useful when debugging large problems. Furthermore, PlanWorks can be used in planner control mode which allows a user to load steps during the planner execution. Each set of steps is called a sequence. Each sequence can be queried and viewed differently depending on the menu options. We will walk you through some of the steps that we have found useful when visualizing EUROPA 2 plans.

For instructions on how to build PlanWorks please refer to the PlanWorks documentation (PlanWorks/README). For detailed instructions on how to operate PlanWorks please refer to PlanWorks/GETTING_STARTED. More detailed instructions on how to visualize your plan can also be found in PLASMA/HELLOWORLD.

Once you have installed PlanWorks, run it using ant. This should bring the initial screen as shown in Figure 3.

You'll see the screen with the following menus: File Project PlugIn and Help (the Windows menu will be disabled since there are no windows inside your PlanWorks workspace). Click on Project and create a new project. Then, select your planning sequence which should be under your plans directory and will be a long name with a sequence number encoded in the name. Figure **??** shows the planning sequence screen.

Once you have loaded a sequence, you'll see two panels: SequenceQuery and SequenceStepsView. The SequenceStepsView shows a histogram of each of the planning steps. The first bar indicates the first step of the plan and the last bar, the last step of the plan. A yellow dot above the bar indicates that data is available for this step. A red dot means that no data is available for this step. The green dot means that the data is available and loaded. The color of the dot changes when you decide to log only some steps or when you use the planner control.

Each bar is composed of three different colors. The green color denotes the number of tokens, the blue color denotes the number of variables, and the sand color denotes the number of constraints. This gives visual feedback as to the

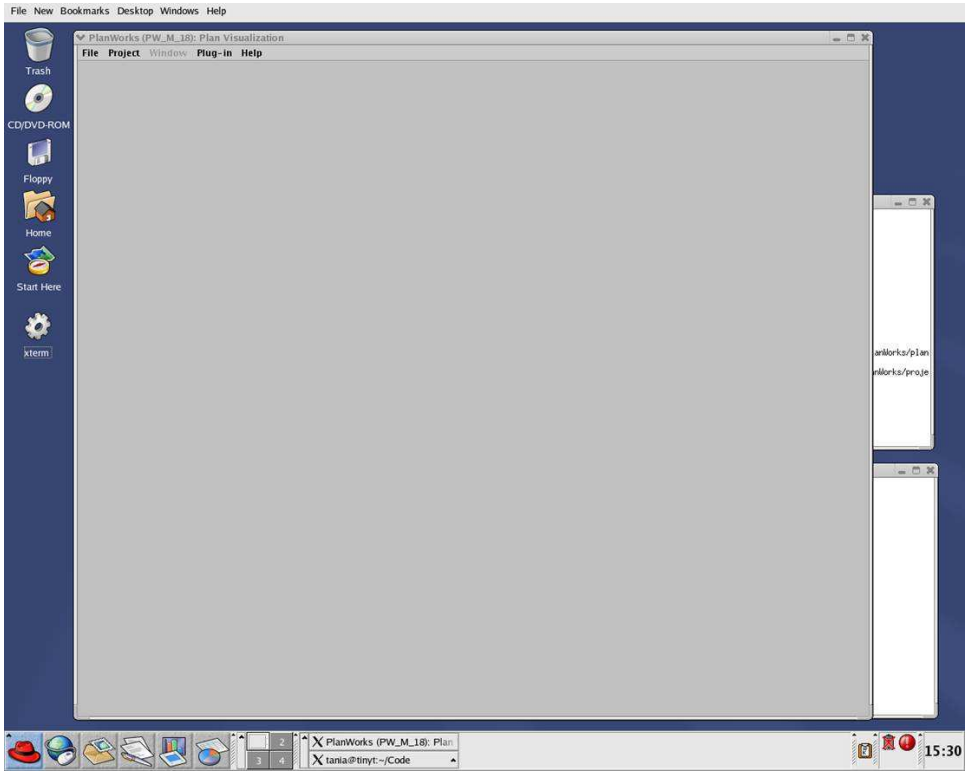

Figure 3: EUROPA 2 PlanWorks Initial Screen

proportion of tokens, variables, and constraints in each step of the plannning process.

Right-click on any column in the step view, you'll see a set of menus: Constraint Network View, DB Transaction View, Decision View, Resource Profile View, Resource Transaction View, Temporal Extent View, Timeline view, and Token Network View as shown in Figure 5.

Now, let's get some information from the plan. Let's focus on a Going token that takes spirit from the lander to rock4. Let's click on the last step and bring up the timeline view. You can get more information about the Going token by right-clicking on it and bringing up the Navigator view. You'll see the token in the center of the view with its variables beneath it and the rule that is responsible for it's existence on the top. You can find the parent token by clicking on the rule. You'll see the token Navigator.At appear in blue. It is blue because that is the color of the Nav timeline. You will also see the other Going token appear which also results from appylying rule2. Figure 6 shows the Timeline and Navigator View.

Since there's too much information and we are looking for the Going token, why don't we limit the view to show only Going tokens. We can do this by

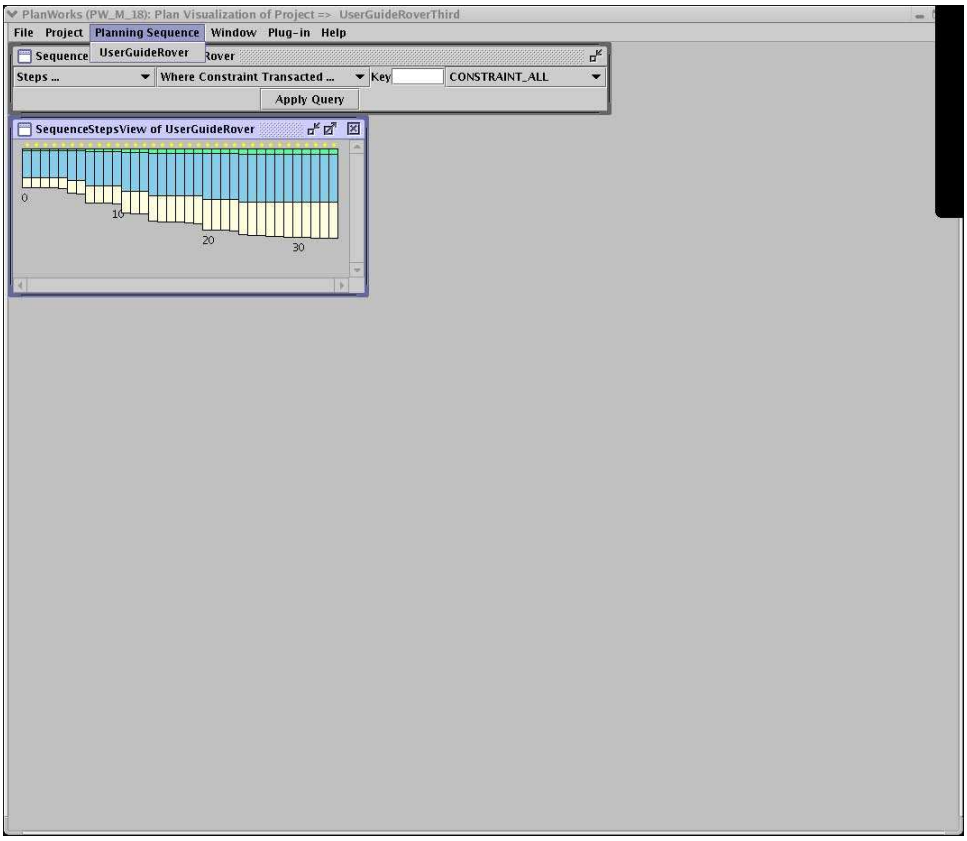

Figure 4: EUROPA 2 PlanWorks Planning Sequence

using the Content Filter. The content filter allows you to restrict the view to only those things that you're interested in viewing. Click on the content filter window. In Predicate, type Navigator.Going. Then press "Apply Filter". You should see the timeline view restricted to the free Going tokens only, with no tokens on any of the timelines. That is because there are currently no inserted tokens on any timeline (nav is the only possible one). Figure 7 shows the timeline view with Going tokens only.

Now, let's identify the Going token that we want to find out more information about. The way to do this is to query for the token events that affected the token. This will give us an idea of what happen throughout the lifetime of that token. To do this, identify token 145 in the spirit.nav timeline. Then bring up the token query results by entering the token key 145 in the Sequence Query window. Identify first the Sequence Query window. It should be long and thin and near the top of your workspace. The options on the window should show: Steps ... Where Constraint Transacted ... Key ... CONSTRAINT_ALL. Instead of "Where Constarint Transacted" select "Where Token Transacted" and enter

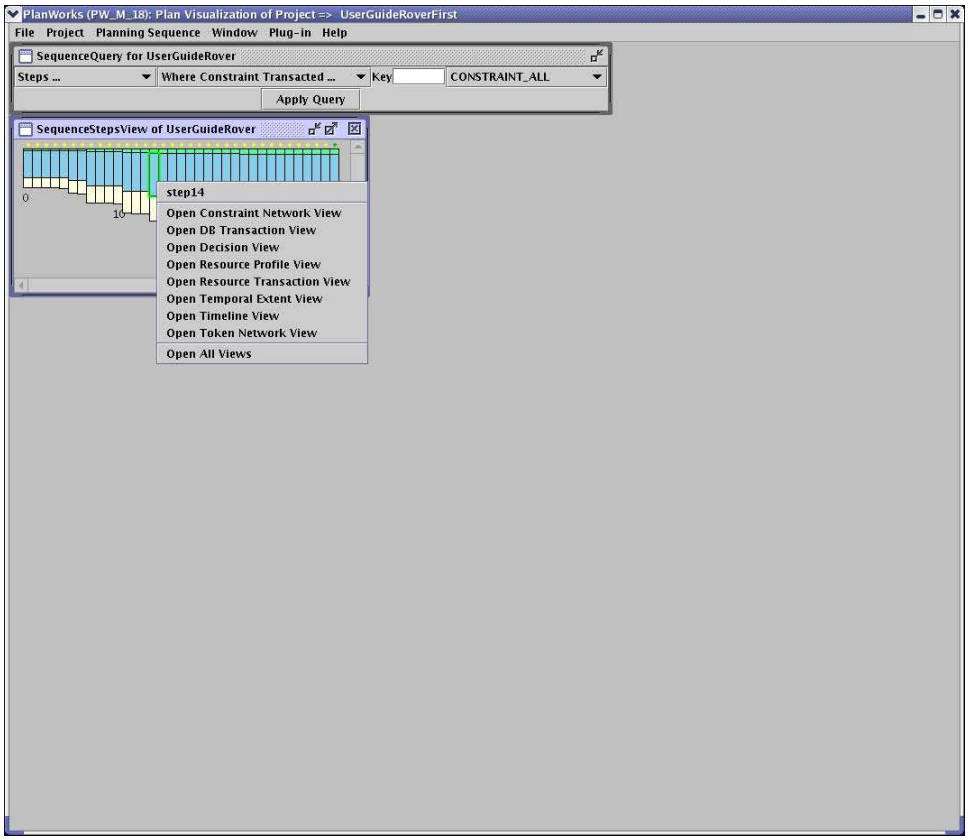

Figure 5: EUROPA 2 PlanWorks Step Menu

the key 145 in the key field, then press Apply Query. This will bring a window with the set of transactions that apply to the Going token 145. Figure 8 shows the results of the transaction query.

Notice that this window shows that the token was created in step 0 and activated in step 14 and inserted in step 15. Notice that in step 14 it is also added to an object. There is only one object that could accept the Going since our model has a single rover so there is a single "Token_Added_to_Object" transaction.

Let's find out next who mandated the creation of the Going token and what subgoal rules are acting upon it. In order to do so, we should bring up the Token Network View. Right-click on Step 14 and select TokenNetwork View. Double click on the Going token to expand and follow the expansion one level up and one level down. This will give you an idea of its parent and its subgoals. Then, right-click on Battery.Change token and select the RuleInstance View. This will bring up the rule code in a separate window. This is useful when you're debugging your model as well as your planner. Figure **??** shows the

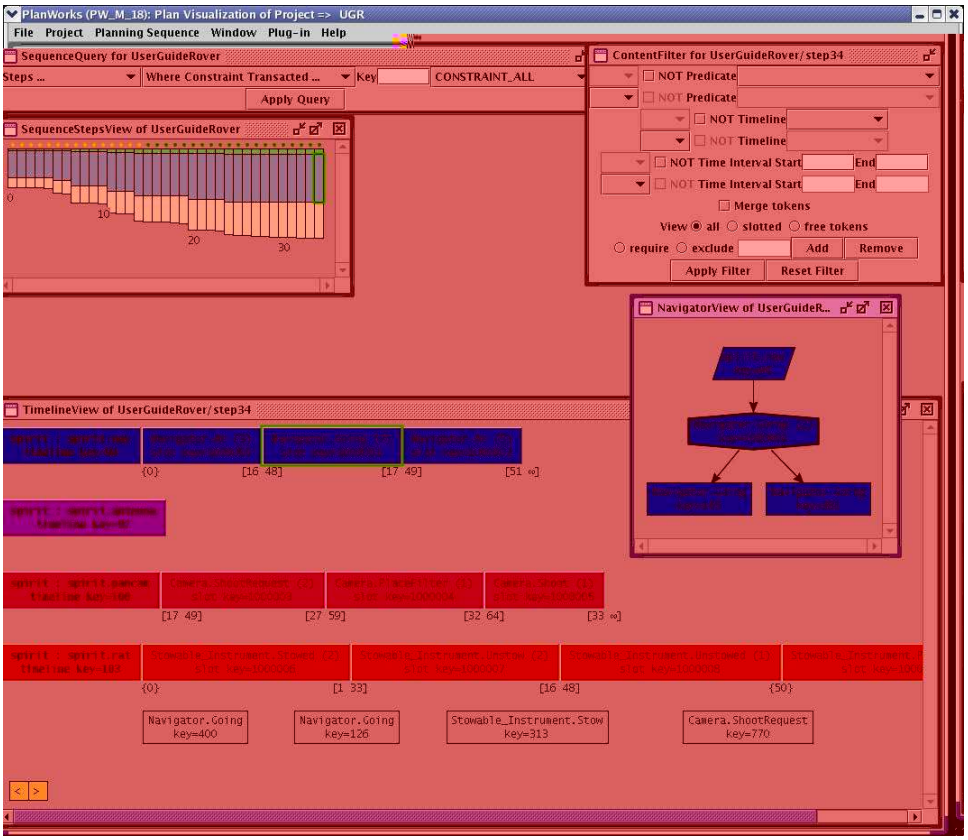

Figure 6: EUROPA 2 PlanWorks Timeline and Navigator View

Token Network View just described.

If you have questions about the resource levels and want to view the resource profile, you can click on Step 34 which should show the effect of the resources on the final plan. The resource profile view shows the resource envelope and the limits imposed. The transaction view shows all transactions to date. Right-click on the transaction in the middle row and bring up the Navigator view to display information about the transaction. You'll see that Rule 8 is responsible for this transaction. Click on Rule 8 and you should see the Going token with key = 145. Figure 9 shows the Resource Profile and Resource Transaction windows for the last step.

Let's open the Constraint Network view and let's see what constraints are acting on the variables of the Going token. Click on every variable from left to right. You'll have to scroll to get to the other variables since the view expands beyond your window. In order to see the entire data, you can bring up an overview window. Right-click on the background of the Constraint Network view. Select the "Overview Window". A small windows showing the entire

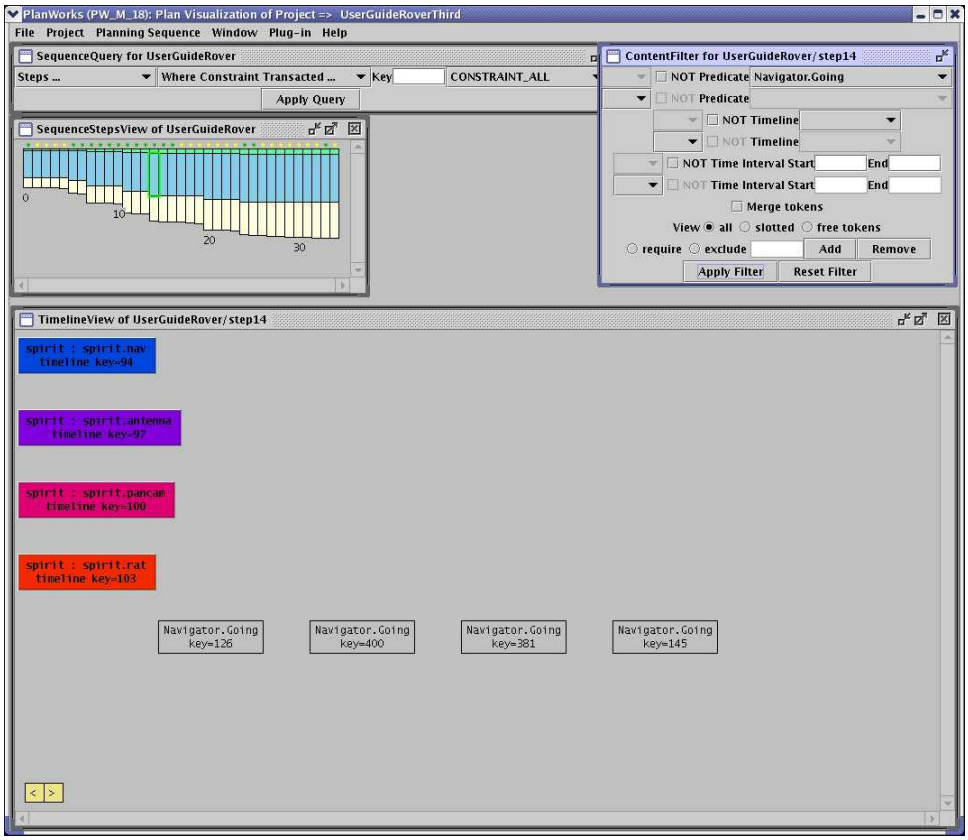

Figure 7: EUROPA 2 PlanWorks Timeline View with Going only

data that can be displayed in this view will pop up. The rectangle shows the data currently in view. The Constraint Network view allows you to view all variables and consraints and their relationships to each other. Figure 10 shows the Constraint Network View with the Overview window.

Let's open the Temporal Extent view on step 15. Scroll down until you see token Going with key = 145. The top arrows that are above the line and pointing down show the range for the start time. The bottom arrows that are beneath the line and pointing up show the range for the end time. Mousing over the arrow will show you the time. In this view you can show the timescale line which will help you view the events that happen before, at and after that time. You can bring this line up by right-clicking on the background of the view and selecting Set Time Scale Line. You should see a red line appear wherever you had your mouse last. That is also the current method to move the time scale line. Figure 11 shows time 35.

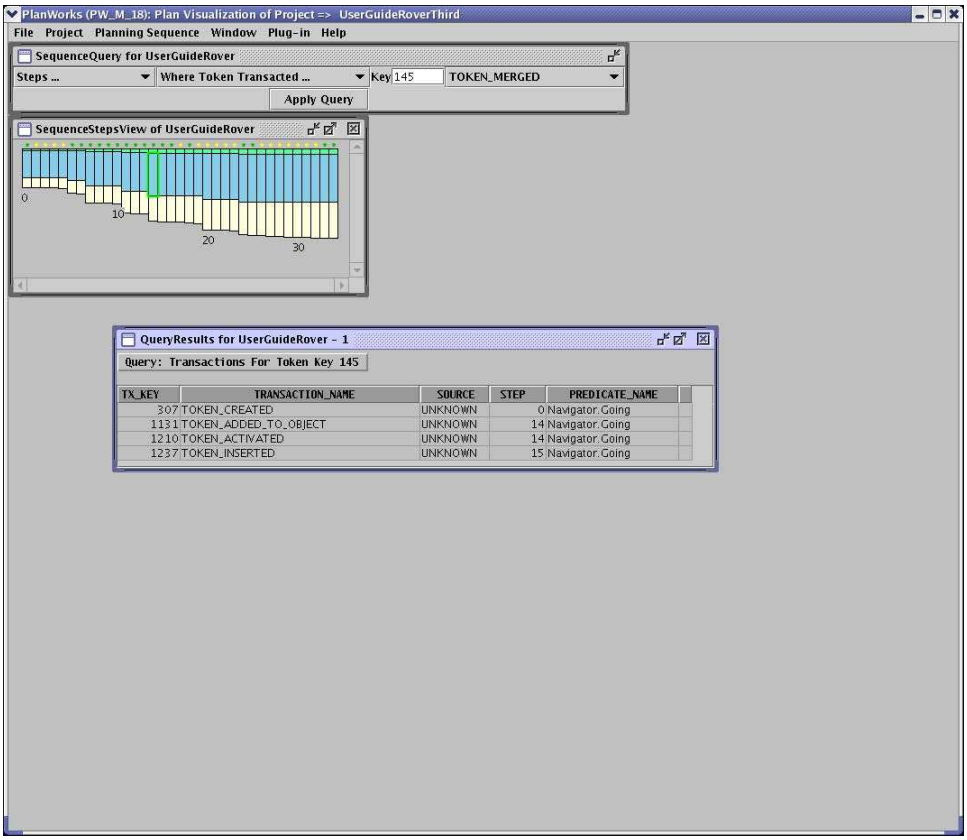

Figure 8: EUROPA 2 PlanWorks Token Query

4.11 Debugging

Another useful tool for debugging your models and planners is logging. We have extensive logging that you can configure through a file called Debug.cfg. This file must exist in the same directory as the executable or the debug information will not be logged. Debug information is by default redirected to std::cout. You can change that by using DebugMsg::setStream().

EUROPA 2 code is instrumented with debug messages of the form:

```
debugMsg("marker", "token " << tokenId << " created");
```

which produce output like this:

```
TokenNetwork.cc:6277: token id_56 created
```

if and only if the marker is found in the Debug.cfg file or debugging has been enabled in code.

The Debug.cfg file format is defined according to the following: ,#,; are line comment characters and their scope ends at the end of line. You can enable

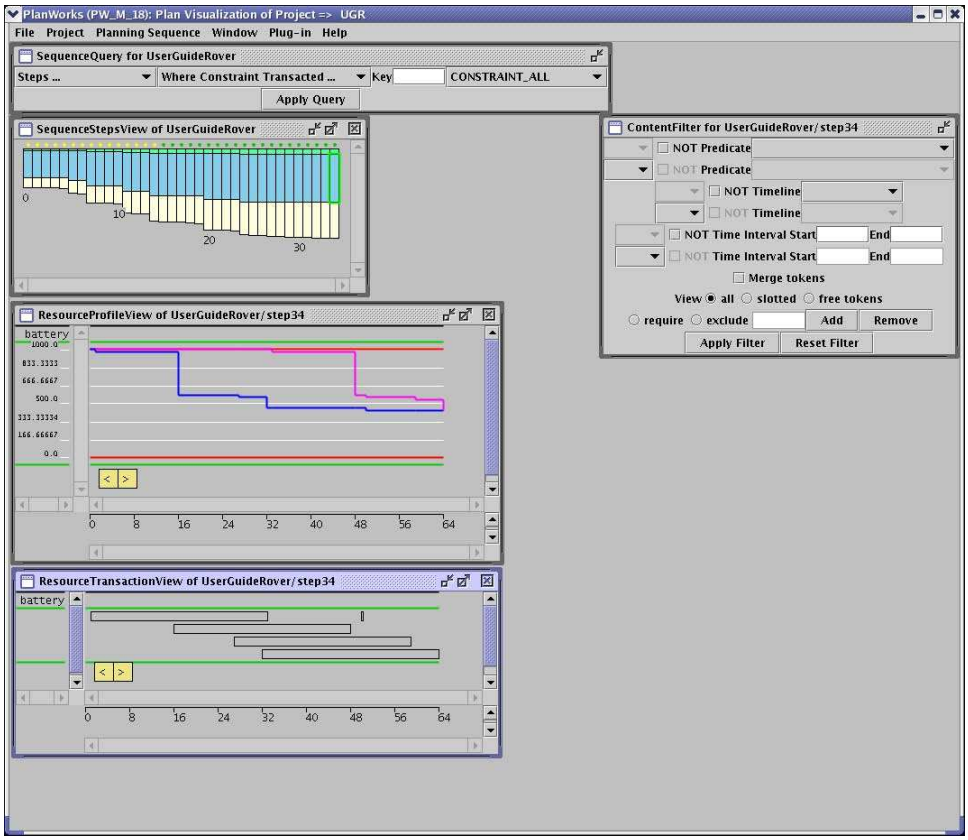

Figure 9: EUROPA 2 PlanWorks Resource Profile and Transaction View

marker debug messages by file or all messages in a file or all messages matching the marker accross all files as in the following examples:

```
file:marker #comment
file /comment
:marker ;comment
```

Each non-comment (and non-empty) line enables all matching debug messages, including any that have the given 'marker' string as any substring of their own marker.

To enable debug messages in code, the stream to write them to must be assigned:

```
DebugMessage::setStream(std::cerr);
```

to send them to std::cerr. All messages can then be enabled with:

```
DebugMessage::enableAll();
```

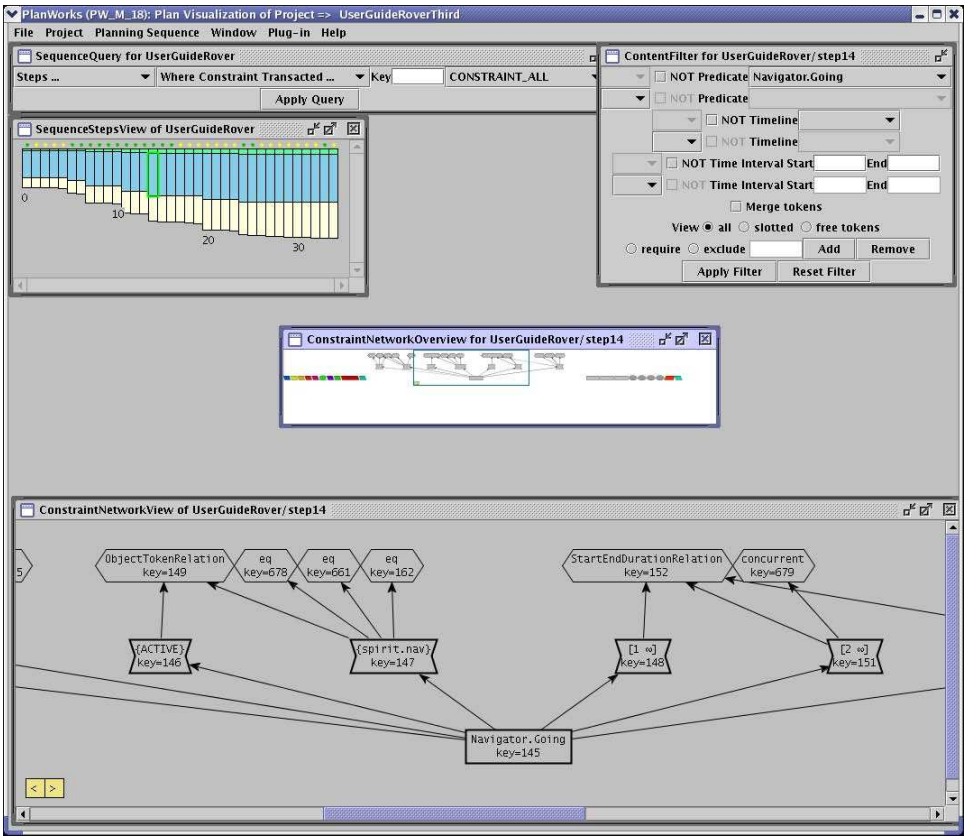

Figure 10: EUROPA 2 PlanWorks Constraint Network View with Overview

and an individual one can be enabled with:

```
DebugMessage *msg;
msg->enable();
```

An individual debug message can be looked up using:

```
msg = DebugMessage::findMsg("file", "marker");
```

If this matches more then one existing debug message, the first one found will be returned. To find all messages in a given file, e.g.:

```
std::list<DebugMessage*> msgs;
DebugMessage::findMatchingMsgs("file", "", &msgs);
```

where the second argument is a empty (zero length) std::string. Note that msgs is not cleared (emptied) by this function, only added to. An empty string can also be passed for the file name, so:

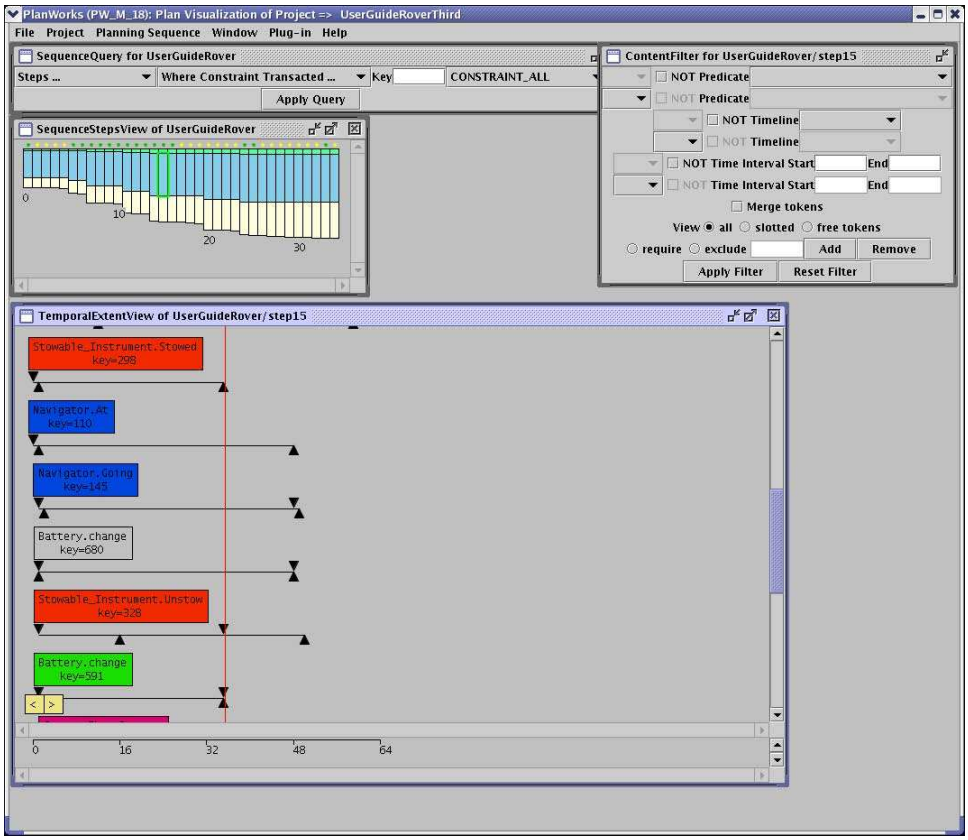

Figure 11: EUROPA 2 PlanWorks Temporal Extent View

```
DebugMessage::findMatchingMsgs("", "", &msgs);
```

will have the same effect as:

```
msgs = DebugMessage::getAllMsgs();
```

except the latter is (currently) a const reference to the internal list and thus runs much faster but cannot be modified.

In all cases, individual messages will not appear in such lists unless the code in question (where the debugMsg() call appears) has already been executed; until then, the info about the individual debug message simply isn't available. Removing this restriction would require the complete list of debug messages to be constructed at compile time (similar to how the entire list of parameter constraint functions is presently done at compile time). However, the calls:

```
DebugMessage::enableAll();
DebugMessage::disableAll();
DebugMessage::readConfigFile(istream& is);
```

are not restricted to existing messages, as they store the "patterns" that are presently enabled and, when a new message is created that matches any of the enabled patterns, it is immediately enabled (and therefore prints its message immediately after being created, as part of the debugMsg() macro). The method:

```
DebugMessage::enableMatchingMessages("file", "marker");
```

adds the appropriate pattern to this internal list of enabled patterns, which is checked immediately for existing debug messages and also when a new debug message is created.

There is no corresponding disableMatchingMessages() in the current implementation, but that could be very tricky (or costly at run time) to implement for cases like:

```
DebugMessage::enableAll();
DebugMessage::disableMatchingMsgs("", "marker");
DebugMessage::disableMatchingMsgs("file", "");
DebugMessage::enableMatchingMsgs("", "marker");
```

since there is no explicit list of files or markers mentioned in debug messages.

Aver is a language for automatically verifying planners, models and plan databases. It allows the description of partial or complete plans and events that occur during planning that constitute expected behavior. An assertion is a boolean statement that examines a particular aspect of a plan (how many "Foo" tokens exist) or the planning behavior (whether or not a "backtrack" message occurred) and asserts something about it. Assertions are preceeded by a specification of when the assertion must be true. They are grouped into tests that can be further organized into super-tests. Files containing collections of tests and assertions are converted into XML and then compiled into Aver instructions which are executed at run-time. See the API documentation for further information on Aver. See also the Aver specification in Section 8.4.

5 PLASMA System Architecture

Figure 12 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,

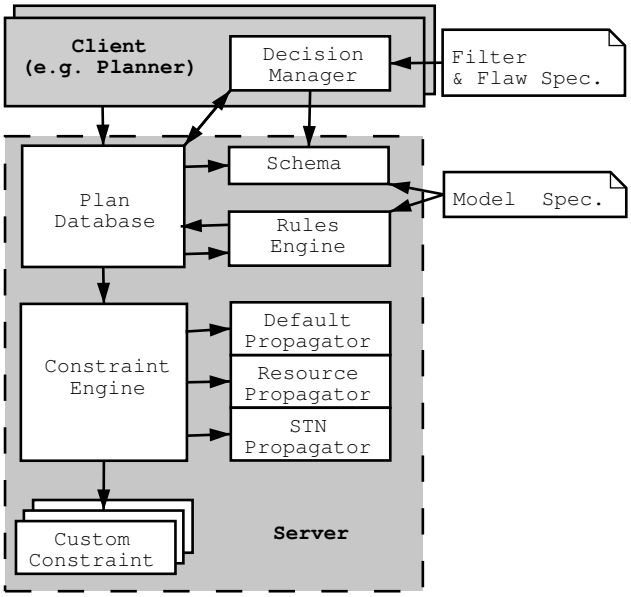

Figure 12: EUROPA 2 System Architecture Diagram

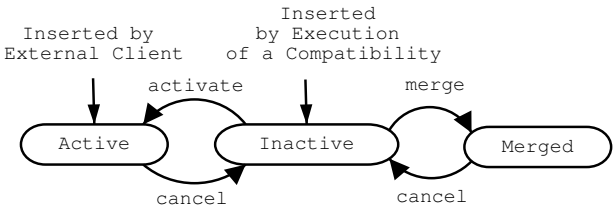

Figure 13: Token State Transition Diagram

though many applications develop their own clients. The *Decision Manager* uses a view specification to manage the set of flaws for a client. Please refer to the API documentation which can be generated by typing `jam documentation` in the top-level directory. To view the API documentation, point your browser to the `documentation/html/index.html` file at the top-level directory.

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

EUROPA 2 provides the capability to pick and choose components and configure them based on application needs. For example, assume that you have an application that requires constraint reasoning services. You may want to use only the Constraint Engine. In this case, you can create your own assembly where you allocate a Constraint Engine and instantiate a problem based on a problem description that you're given, or using the ConstraintEngine, Constraint and Variable APIs directly. You can similarly construct an assembly with a PlanDatabase only or a PlanDatabase and a planner. If your application doesn't require resource reasoning you don't have to include resources in your assembly. There are a few examples in the code where we create specialized assemblies depending on the use we have. An example is StandardAssembly which provides an example of an assembly that uses resource reasoning, a plan database, a constraint engine with a specialized (STN) temporal propagator, and a chronological backtracking planner.

6.2 Using and extending the CBPlanner

6.3 Custom constraints

Let's say that you now require special constraints and have available a propagation algorithm that knows and handles those constraints. Assume that you now want to handle more general classes of constraints. You can use the Constraint Engine provided with PLASMA to handle the more general classes of constraints and configure it to use the special-purpose propagation algorithm you have to handle the types of constraints you already can handle. In order to do this, you must create a propagator wrapper on your propagation algorithm (we'll explain how to do this in the next section). You must then register this propagator with the Constraint Engine. Then, you must register the types of constraints with the Constraint Engine and the corresponding propagator. This information will be captured by the ConstraintEngine who will route the constraint propagation request to the appropriate propagator. All that you have to ensure is that the propagator is registered with the ConstraintEngine and that whenever you want to create a new constraint instance, the type of constraint has been registered.

6.4 Custom propagation

How do we write custom propagation? Currently, EUROPA 2 provides a single way to configure custom propagation and that is through the use of propagators. A propagator is a core interface that provides the capability to manage an agenda of constraints. A propagator provides query interfaces that provide status, enable execution, allow registration of constraints and appropriate linkage to the Constraint Engine. We currently provide a TemporalPropagator and a ResourcePropagator.

6.5 Building model specializations

Here we talk about extending the object model.

6.6 Custom rule implementations

Here we talk about writing custom rules and we highlight that there's no mechanism to hook these rules up to NDDL.

6.7 Specialized domains

Here we discuss the type factory.

6.8 External data integration

6.9 Listeners and Loggers

6.10 Integration to PlanWorks

We discuss in this section how we integrate with PlanWorks. We mention the PlanWorks.cfg and the PlannerControl object and how PlanWorks can be used to query the sql database.

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

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

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

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

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

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

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

[8] ILOG. Ilog solver: User manual, July 1996. Version 3.2.

[9] I.A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin, and Won Soo Kim. Claraty: An architecture for reusable robotic software. In *Proceedings of the SPIE Aerosense Conference*, 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 Relation
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 +Inf] [1 +Inf]
A inverse-during B (id)	A.start ¡ B.start, A.end ¿ B.end	A contains [1 +Inf] [1 +Inf] B
A starts B (s)	A.start = B.start, A.end ¡ B.end	A parallels [0 0] [-Inf -1] B
A inverse-starts B (is)	A.start = B.start, A.end ¿ B.end	A parallels [0 0] [1 +Inf] B
A finishes B (f)	A.start ¿ B.start, A.end = B.end	A parallels [-Inf -1] [0 0] B
A inverse-finishes B (if)	A.start ¡ B.start, A.end = B.end	A parallels [1 +Inf] [0 0] B
A overlaps B (o)	A.start ¡ B.start, A.end ¿ B.start, A.end ¡ B.end	¡unsupported as a single relation
A inverse-overlaps B (io)	A.start ¡ B.end, A.end ¿ B.start, A.end ¿ B.end	¡unsupported as a single relation

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
A inverse-overlaps B (io)	LessThan(A.start, B.end); LessThan(B.start, A.end); LessThan(B.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.