HATP
User Documentation

September 2012

by

J. GUITTON

julien.guitton@laas.fr
julio38@gmail.com

Date	Author	Comment
April 2013	Raphaël Lallement raphael.lallement@laas.fr	Add information about default value for every attribute type.

Table of Contents

1 Introduction....3
1.1 Overview of the document....3
1.2 Human-Aware Task Planner....3
1.2.1 Task planning....3
1.2.2 HTN planning....3
1.2.3 Agents and action streams....4
1.2.4 Action costs and social rules....4
2 The HATP formalism....5
2.1 Entities and attributes....5
2.1.1 Static vs dynamic attributes....5
2.1.2 Atom or set attribute types....5
2.1.3 Declaration of an entity 5
2.2 Actions and Methods....6
2.2.1 Actions....6
2.2.2 Methods....6
2.3 Operations in preconditions and effects....7
2.3.1 Operations in preconditions....7
2.3.2 Operations in effects....8
2.4 Logical expressions....8
2.4.1 Conjunction....8
2.4.2 Disjunction....8
2.4.3 Negation....9
2.5 Variable assignment / substitution....9
2.5.1 Free variable....9
2.5.2 First satisfier precondition....10
2.5.3 Sorted precondition....10
2.6 Quantification....10
2.6.1 Universal quantifier....10
2.6.2 Existential quantifier....11
2.7 Call term and conditional effect....11
2.7.1 Call term....11
2.7.2 Conditional effect....11
2.8 Cost, duration and social rules....12
2.8.1 Cost and duration functions....12
2.8.2 Social rules....13
2.8.3 Setting up costs....15
2.9 Planning domain and problem....15
2.9.1 Organization of the planning domain file....15
2.9.2 Defining a problem....16
2.10 HATP Extension: Belief Management....17
2.10.1 The agents: myself and the others....17
2.10.2 Beliefs representation....17
2.10.3 Known and unknown information....18
2.10.4 Communication actions....18
2.10.5 General communication method....19

3 Installing HATP....20
3.1 HATPOnboard....20
3.2 MsgConnector....20
3.3 HATPConsole....21
3.4 LibHATP....22
4 Running HATP....23
4.1 How works HATP ?....23
4.1.1 Overview of the planning system....23
4.1.2 Planning process....23
4.1.3 Important files of HATPOnboard....24
4.2 Launching the system....24
4.3 Communication with HATP....25
4.3.1 Communication API....25
4.3.2 Basic communication syntax....26
4.3.3 Requests syntax....26
4.3.4 Answer of HATP....27
4.4 Manipulation of plans....28
4.4.1 LibHATP....28
4.4.2 A simple application....29
A The “Clean The Table” domain....31
B The “Dock Worker Robot” domain....35

1 Introduction

This document aims at presenting the basis of HATP: How to use it, how to write a planning domain, how to launch the planner and the attached third part modules, how to manipulate a plan produced by HATP...

1.1 Overview of the document

This document is divided in three parts. The first part describes the HATP planning language and explains how to write a planning domain. The second part presents the different modules composing the HATP framework and how to install them. The last part explains how to use HATP and how to manipulate a plan produced by the planner.

1.2 Human-Aware Task Planner

HATP stands for Human-Aware Task Planner. It has been initiated during the PhD of Vincent Montreuil[1] then completed during the PhD of Samir Alili[2]. Finally I have debugged it and added some useful abilities between 2010 and 2012.

HATP is a centralized multi-agent planner. It is able to produce plans simultaneously for all the involved agents. It can be tuned by setting up different costs depending on the actions to apply and by taking into account a set of global constraints called social rules. This tuning aims at adapting the agent's behaviors according to some preferences and to the desired level of cooperation between the agents.

1.2.1 Task planning

Task planning is a symbolic reasoning aiming at construction a plan of actions, allowing to reach a goal state starting from an initial state. Actions are instances of operators defined by a set of preconditions and a set of effects. Preconditions are conditions that must be true in the current state in order to apply the action. Effects model the changes in the environment resulting from the achievement of the chosen action.

1.2.2 HTN planning

HATP is a HTN planner. The aim of Hierarchical Task Planning is to decompose a high-level task representing the goal to achieve into a set of sub-tasks until reaching atomic tasks that are achievable by the agents.

One of the most well-known HTN planning is SHOP2. Problems solved with SHOP2 can be translated quite easily into the HATP formalism and solved in a time quite similar to the time

1 V. Montreuil - “Interaction Décisionnelle homme-robot : la planification de tâches au service de la sociabilité du robot ”, 2008

2 S. Alili - “Interaction Décisionnelle Homme-Robot : Planification de tâche pour un robot interactif en environnement humain”, 2010

taken by SHOP2.

1.2.3 Agents and action streams

The planner produces actions for all the agents involved in the problem. The resulting plan, called “shared plan”, is a set of actions that forms a stream for each agent. Depending on the context, some shared plans contain causal relations between the agent's streams. For example, the second agent needs to wait for the success of the first agent's action. When the plan is performed, these causal links induce some synchronizations between agents.

1.2.4 Action costs and social rules

To each action is associated a cost function and a duration function. The duration function provides a duration interval for the action achievement and is used, on the one hand, as a timeline to schedule the different streams and, on the other hand, as an additional cost function.

In addition to these costs, HATP takes as an entry a set of global rules called social rules. Social rules are constraints aiming at leading the plan construction towards the best plan according to some preferences. The main rules are:

- Undesirable state
- Undesirable sequence
- Effort balancing
- Wasted time
- Control of intricacy
- Bad decomposition

2 The HATP formalism

Classically in automated symbolic planning, the facts describing the world state are defined in a formalism based on the first order logic (e.g. PDDL). HATP uses an object-oriented modeling language in which each element of the environment is defined as a distinct object. This language has nearly the same expressive power and features than SHOP2.

2.1 Entities and attributes

An object is called *entity* and its characteristics are defined by a set of *attributes*. Each entity is unique but refers to a class or type of entity regrouping the objects of same type. This type is characterized by a name and a set of attributes.

2.1.1 Static vs dynamic attributes

Attributes are defined to be *static* or *dynamic*. A static attribute represents a non-modifiable information whereas a dynamic attribute can be updated during the search.

2.1.2 Atom or set attribute types

An attribute can be an *atom* and then can take on value at a time or it can be a *set* and in this case it is used to store multiple values. These values have a type than can be an entity or a predefined type: string, bool or number.

The default value for each type is:

Type	Value
bool	false
number	0.0

Type	Value
string	“”
Entity	NULL

2.1.3 Declaration of an entity

The type of an entity is defined using the keywords `define entityType` followed by the name of the entity type:

```
define entityType Location;
```

Then, for each entity type, the corresponding set of attributes is defined using the keywords `define entityAttributes`. For example:

```
define entityAttributes Agent {
    static atom string type;
    dynamic atom Location at;
    dynamic atom Container loaded;
}
define entityAttributes Location {
    static set Location adjacent;
    dynamic atom bool occupied;
    dynamic atom Agent isForbiddenBy;
}
```

The entity type `Agent` is mandatory and predefined. But the specification of its attributes is free.

2.2 Actions and Methods

The aim of HTN planning is to decompose tasks of high level into sub-tasks until reaching primitive tasks that cannot be decomposed. High level tasks are described through a structure called method and primitive tasks are represented by actions.

2.2.1 Actions

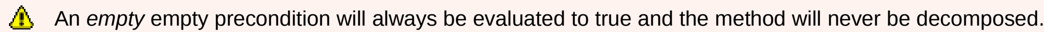
Actions are defined by a set of preconditions, a set of effects, a reference to a cost function and a reference to a duration function:

```
action Move(Agent R, Location From, Location To) {
    preconditions {
        R.type == "ROBOT";
        To >> From.adjacent;
        R.at == From;
        To.occupied == false;
    };
    effects {
        R.at = To;
        From.occupied = false;
        To.occupied = true;
    };
    cost{costFn(1)};
    duration{durationFn(1, 1)};
}
```

2.2.2 Methods

A method is described by a precondition called empty condition and a set of decomposition. This empty condition aims at providing a stop point in the development of the current branch of the planning tree. A decomposition models one way to decompose the current task into sub-tasks and is defined by a set of preconditions and a set of sub-tasks:

```
method Navigate(Agent R, Location To) {
    empty {R.at == To;};
    {
        preconditions {};
        subtasks {
            From = SELECT(Location, {R.at == From;});
            1: NavFromTo(R, From, To);
        };
    }
    {
        // another decomposition...
    }
}
```


An *empty* empty precondition will always be evaluated to true and the method will never be decomposed.

The sub-tasks can be partially or totally ordered. In the last case, the order must be explicit and is encoded by the symbol `>` followed by the numeric Id of the previous task, after the name of a sub-tasks : *(you can specify several sub-tasks, see last line in next example)*

```
subtasks {
    1: Take(G, K, SourcePile);
    2: Put(G, K, TargetPile)>1;
    3: Pull(G, K, SourcePile);
    4: TakeAndPutAll(SourcePile, TargetPile)>2,>3;
};
```

2.3 Operations in preconditions and effects

Manipulation of entities and attributes are done with specific operators. In this section, we present these operators allowing to compare values in preconditions and to update them in effects.

2.3.1 Operations in preconditions

Comparison operators can be regrouped into two class: operators for testing values of atoms and operators for testing values of set.

Atom testing operators

To test if an atom is equal to a value, the operator `==` is used. The operator `!=` stands for different. The left part and the right part of the comparison expression must be of the same type: a number with a number, a string with a string, a boolean with a boolean and an entity with an entity.

```
// R1 and R2 are two entities of type Agent
R1 != R2;           // comparison of entities
R1.fuel == 100;     // comparison of numbers
R1.type == "ROBOT"; // comparison of strings
R1.type == R2.type; // comparison of strings
R1.isReady == true; // comparison of booleans
R1 != null          // comparison of entities (existence)
```

For booleans, the keywords `true` and `false` represent the corresponding values. To test if a variable of type entity is declared, the keyword `NULL` is used.

Numerical comparisons are done with the operators `<`, `<=`, `>`, `>=` (inferior, inferior or equal, superior, superior or equal).

```
R1.fuel < 100;
R1.fuel >= 100;
```

Set testing operators

The operator >> is used to test if a value is a member of a set and `!>>` if the value is absent of this set. To get the number of elements in a set, we can use the keyword `.size()`

```
// R1, R2 and R3 are entities of type Agent
R1.friends.size() > 0;  // R1 should have at least one friend
R2 >> R1.friends;       // R2 should be a friend of R1
R3 !>> R1.friends;      // R3 must not be a friend of R1
```

2.3.2 Operations in effects

In the same way, we have defined modification operators for atoms and modification operators for sets.

Atom modification

The assignment of a new value to an atom is done through the operator =. Left and right part of the expression must have the same type.

```
R1.type = "ROBOT";
R1.fuel = 100;
R1.isReady = true;
```

Set modification

To insert an element into a set, the operator is <<= and to remove an element from a set, the operator is =>>. The inserted or deleted element must be of the same type than the set.

```
R1.friends <<= R2;      // R2 is now a friend of R1
R1.friends =>> R2;      // R2 is not anymore a friend of R1
```

When inserting an element of type entity, only a reference to this entity is inserted. When inserting twice a same value, the set is not changed.

```
R1.friends <<= R2;
R1.friends <<= R2;
// R1.friends.size() == 1
```

2.4 Logical expressions

In the first order logic, predicates in preconditions can be of the form: conjunction, disjunction or negation. HATP don't use the first order logic but the formalism is similar.

2.4.1 Conjunction

By default, the set of preconditions forms an implicit conjunction. For example :

```
R.type == "CAR";    // and
R.fuel == 100;
```

2.4.2 Disjunction

In HATP, it is possible to use disjunctions when defining preconditions. These disjunction must be explicit and are applied to two conjunctions:

```
OR {
    R.type == "CAR";
    R.fuel == 100;
}{
    R.type == "TRUCK";
    R.fuel == 150;
}
```

In this example, the preconditions are verified if R is a car and has 100 units of fuel or if R is a truck and has 150 units of fuel.

2.4.3 Negation

There is no explicit negation in HATP. Negations are done through the different comparison operators. For example:

```
R.type != "CAR";    // R is not a car
R.isReady == false; // R is not ready
R2 !>> R.friends;   // R2 is not a friend of R
```

2.5 Variable assignment / substitution

During the planning process, variables of preconditions and effects are substituted by the values of the current task parameters. These values are inherited from the parent task. However, HATP has a mechanism to assign free variables. This mechanism takes place in the decomposition just before the description of the sub-tasks list.

```
method Navigate(Agent R, Location To) {
    empty {R.at == To;};
    {
        preconditions {};
        subtasks {
            // here need to assign the free variable From
            1: NavFromTo(R, From, To);
        };
    }
}
```

2.5.1 Free variable

The assignment of free variable can be done only in a decomposition using the function `SELECT` that takes as parameters a type and a set of preconditions. For example:

```
// Select an agent of type robot and assign it to the variable R
R = SELECT(Agent, {R.type == “ROBOT”;});
// Select the location where this agent is currently
From = SELECT(Location, {R.at == From;});
```

2.5.2 First satisfier precondition

This possibility to use free variables induces a branching in the planning tree. Sometimes, when several values are possible and when the choice has no incidence on the quality of the plan, or to reduce the planning time, we would like that the planner uses the first value it has found.

In other words, a first satisfier precondition causes the planner to consider only the first binding. Alternative bindings will not be considered event if the first binding leads to an invalid plan. The function is named `SELECTONCE`.

```
// Select the FIRST agent FOUND of type robot and assign it to the variable R
R = SELECTONCE(Agent, {R.type == “ROBOT”;}); // do not backtrack on this choice
```

2.5.3 Sorted precondition

A sorted precondition causes to consider bindings for this precondition in a specific order. The associated function is named `SELECTORDERED` and takes two additional parameters a comparison function and an comparison order.

```
// Select the agent which is the the closer from the location From
R = SELECTORDERED(Agent, {R.type == "ROBOT";}, distanceMin(R.at, From), <);
```

The comparison function is a cost function (see section 2.8) and should be defined in the same way.

2.6 Quantification

The HATP formalism allows to express universal quantifier and existential quantifier. These quantifiers can be used either in preconditions or effects.

2.6.1 Universal quantifier

The universal quantifier means “for all” and is expressed in HATP with the keyword `FORALL`. Used in preconditions, the universal quantifier meaning is “for each possible substitution of the variable of the specified type of entity, if it respects the first set of precondition then it must

respect the second set of preconditions:

```
// All the agent that are at From must be ready
FORALL(Agent R, {R.at == From;}, {R.isReady == true;});
```

In this example, for all agents `R` positioned at the location `From`, `R` must be ready.

The universal quantifier is used in effects to manipulate a set of entities that satisfies a given set of preconditions and apply to these entities some specified effects.

```
// set isReady to true for all the agent that are at From
FORALL(Agent R, {R.at == From;}, {R.isReady = true;});
```

2.6.2 Existential quantifier

The existential quantifier is expressed with the keyword `EXIST` in HATP. This quantifier can only be used in preconditions and is expressed with two set of preconditions. The first set allows to pre-select a set of entities and the second to test if the preconditions are true for at least one of these entities.

```
// At least one agent that is at From must be ready
EXIST(Agent R, {R.at == From;}, {R.isReady == true;});
```

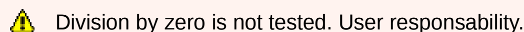
2.7 Call term and conditional effect

The last two functionalities of the HATP domain formalism are the *call term* and the *conditional effect*.

2.7.1 Call term

The keyword `CALL` is used to apply mathematical operations (`+`, `-`, `*`, `/`) over a variable. The first variable of the function receives implicitly the resulting value. A call term can be used only in the effects.

```
// remove 10 units of fuel to R
CALL(R.fuel - 10); // meaning: R.fuel = R.fuel – 10;
```


Division by zero is not tested. User responsability.

2.7.2 Conditional effect

HATP allows to express some conditional effects. The associated keyword is `IF` and the

formalism is `IF{set of preconditions;}{set of effects}`.

```
// if the fuel level is less than 5 units then R is not ready anymore
// and increment the number of failures of R
IF{R.isReady == true; R.fuel < 5;}{R.isReady = false; CALL(R.failures + 1);}
```

2.8 Cost, duration and social rules

In this paragraph, we explain how to write cost and duration functions, and how to write and use social rules.

2.8.1 Cost and duration functions

Cost and duration functions aim at defining a cost and a duration to each primitive task, i.e., to each action. They are linked to the action through the cost and duration fields of the action description:

```
action Move(Agent R, Location From, Location To) {
    preconditions { ...};
    effects { ...};
    cost{costFn(1)};                // link to the costFn cost function
    duration{durationFn(1, 1)}; // link to the durationFn duration function
}
```

Cost and duration functions are C++ function written in the cost file. A cost function must return a `double` and a duration function must return a `pair` of `double`. The arguments of cost and duration function can be values with standard type of HATP (`number`, `string` or `bool`) or entities.

```
// example of a cost function
double costFn(number x){
    return (double)x;
}
// example of a duration function
pair<double, double> durationFn(number x1, number x2){
    return make_pair((double)x1, (double)x2);
}
```

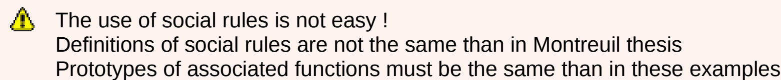
For entities, the C++ type is `LightEntity`. The bindings between planning values and C++ variables are automatically done during the planning process. In the C++ functions, the entities and their attributes can be manipulated using the function `get<type>("name")` where `type` is the type of the manipulated attribute and `name` the name of this attribute as defined in the domain. *(N.B. see the planning/LightEntity.hh file to see how to get more informations on the entity.)*

```
    // a cost function computing the euclidean distance between 2 entities
    double costDistance(LightEntity A1, LightEntity A2){
        double XA1 = (double)A1.get<number>("xCoordinate");
        double XA2 = (double)A2.get<number>("xCoordinate");
        double YA1 = (double)A1.get<number>("yCoordinate");
        double YA2 = (double)A2.get<number>("yCoordinate");
        double R = ceil(sqrt(pow((XA2-XA1),2) + pow((YA2-YA1),2)));
        return R;
    }
```

2.8.2 Social rules

Social rules aims at providing some extra global criteria when comparing some plans. These rules are applied to a complete plan. Social rules have been defined to be used in the context of human-robot interaction but some of them are generic and can be used in other domains.

These rules are: wasted time, effort balancing, control of intricacy and undesirable sequence. Two additional rules have been defined: undesirable state and bad decomposition, but I have deactivated them due to some bugs (never corrected) in the HATP code.



Wasted time

The wasted time social rule is used to avoid plans in which an agent must wait before executing its action or has inactive periods. The time periods that can be used are: at the beginning of the plan (`double getFromStartDuration()`), at the end of the plan (`double getToEndDuration()`) or between two actions (`vector<double> getDurations()`). To use this social rule, the definition is:

```
    wastedTime {
        priority = 0;                // priority of the social rule
        agents = {ROB1, ROB2};       // list of concerned agents
        penalty{computeWastedTime}; // associated cost function
    }
```

An example of associated cost function for the wasted time social rule is:

```
    // penality = wait time between actions + 10 * wait time at the end
    float computeWastedTime(vector<WastedTimeData*> const& vec, double totalDuration) {
        double inactive = 0.;
        for(unsigned int it = 0 ; it < vec.size() ; it++) {
            vector<double>vecDurations = vec[it]->getDurations(); // for one agent
            for(unsigned int j = 0 ; j < vecDurations.size() ; j++) {
                inactiveDuration += vecDurations[j];
            }
            inactiveDuration += vec[it]->getToEndDuration() * 10;
        }
        return inactiveDuration;
}
```

Effort balancing

The effort balancing social rule aims at controlling the engaged efforts of concerned agents in

the produced plan.

```
effortBalancing {
    priority = 0;                    // priority of the social rule
    agents = {Achile, Jido};            // list of concerned agents
    penalty{computeEffortBalancing};    // associated cost function
}
```

An example of associated cost function for the effort balancing social rule is:

```
// if human works more than robot then cost is 100
float computeEffortBalancing(map<LightEntity const*, float> efforts) {
    float penH = 0, penR = 0;
    for(map<LightEntity const*, float>::const_iterator it=efforts.begin() ; it!=efforts.end() ; it++) {
        if ((it->first)->get<string>("type") == "human") penH += it->second;
        else penR += it->second;
    }
    if(penH > penR) return 100;
    return 0.;
}
```

Control of intricacy

The Control of intricacy social rule can be defined to limit dependencies between the actions of two or more agents by controlling the causal links between the streams of these agents. The definition is

```
controlOfIntricacy {
    priority = 0;                       // priority of the social rule
    agents = {ROB1, ROB2, ROB3};        // list of concerned agents
    penalty{computeControlOfIntricacy}; // associated cost function
}
```

An example of associated cost function for the control of intricacy social rule is:

```
// add a penalty of 10 to each causal link
float computeControlOfIntricacy(unsigned int nb) {
    return (float)(nb * 10.);
}
```

Undesirable sequence

The last social rule is the undesirable sequence social rule and is used to penalize some possible decompositions. For example, in the Clean the Table domain, if we prefer that the robot makes accessible an object instead of giving it directly, we can define the rule like this:

```
    undesirableSequence putAndTake {
        priority = 0;
        // definition of variables
        Agent A1, A2;
        GameArtifact C;
        Table T;
        // some preconditions
        conditions {A1 != A2;}
        // undesirable sequence
        sequence {
            1:PutObject(A1, C, T);
            2:TakeObject(A2, C, P2)>1;
        }
        penalty{undesirableSeqPenalty};
}
```

This rule has a name, here `putAndTake`. So we can define, for a same domain, more than one undesirable sequence. An example of associated cost function for the undesirable sequence social rule can be:

```
// penalize each undesirable sequence by 50 * # of time it appears
float undesirableSeqPenalty(vector<SocialRuleData*> const& vec, LightWorldBase const& wb){
    float penalty = 0.;
    for(unsigned int i = 0 ; i < vec.size() ; i++) penalty += 50.*(vec[i]->getNumber());
    return penalty;
}
```

To resume, prototypes of cost functions for social rules are:

Wasted time:	`float name(vector<WastedTimeData*> const&, double)`
Effort balancing:	`float name(map<LightEntity const*, float> )`
Control of intricacy:	`float name(unsigned int)`
Undesirable sequence:	`float name(vector<SocialRuleData*> const&, LightWorldBase const&)`

2.8.3 Setting up costs

As we have seen in the previous paragraph, social rules have a field called `priority`. This priority aims at giving a weight to each social rule as well as to the duration cost compared to the action costs (priority equal to 0 by default). The computation method used is called AHP for Analytic Hierarchy Process[3].

A cost criterion with a priority greater than 0 will be considered as more important than others by HATP. In the opposite, a priority lesser than 0 will be less important. If all the criteria have a priority of 0 than they are equal. For example, with only action costs and duration:

```
// setting up duration priority compared to action costs
timePart { priority = 0; }  // duration and action costs have the same weight = .5
timePart { priority = 2; }  // duration weight is twice action costs weight
timePart { priority = -2; } // action costs weight is twice duration weight
```

Priorities are defined relatively to each other in the interval [-8, 8].

2.9 Planning domain and problem

HATP takes in entry two files: a domain file containing the description of the manipulated objects and the possible tasks, and a cost file containing all the cost functions (action costs, durations and social rules costs).

2.9.1 Organization of the planning domain file

The planning domain file is composed of three parts: the fact database, the HTN part and the social rules part.

3 See for example: http://en.wikipedia.org/wiki/Analytic_hierarchy_process

```
// The domain file:
factdatabase {
    ...
}
HTN {
    ...
}
... // rules part
```

Fact database

The fact database is defined four steps: definition of entity types, definition of attributes, creation of entities and initialization of attributes. This last step is optional and can be done at the beginning of the planning process by an external module as, for example, the supervisor.

```
Factdatabase {
    // step 1: definition of entity types (Agent already defined)
    define entityType Location;
    // step 2: definition of attributes
    define entityAttributes Agent {
        static atom string type;
        dynamic atom Location at;
        dynamic atom Container loaded;
    }
    define entityAttributes Location {
        static set Location adjacent;
        dynamic atom bool occupied;
        dynamic atom Agent isForbiddenBy;
    }
    // step 3: creation of entities
    ROB1 = new Agent;
    LOC1 = new Location
    // step 4: (opt.) init of attributes
    ROB1.type = "ROBOT";
    ROB1.at = LOC1;
    ...
}
```

HTN part

The HTN part contains the list of possible tasks: methods and actions.

Rules part

The end of the file is dedicated to the definition of social rules. The part is optional: A domain can be defined without any social rules. But it must contain the declaration of the priority of the duration criterion.

```
// last part: social rules
timePart { priority = 0; }

wastedTime {
    priority = -1;
    agents = {ROB1, ROB2};
    penalty{computeWastedTime};
}
```

2.9.2 Defining a problem

Because HATP was transformed to be executed onboard a robot, there is no file dedicated to

describe a problem. A goal is sent to HATP by a external module under the form of a task to decompose. Complex goal can be designed as a method in the HTN part.

2.10 HATP Extension: Belief Management

After the CHRIS project, we went to the following conclusion: When acting in an environment with other partners, an agent has to reason not only on its own capabilities but also on the capabilities of other agents in order to achieve a task in a collaborative way.

The description of the world as it can be modeled with HATP assumes that the current state is entirely known and that the agents share the same view of the world. In HRI problems, these assertions can lead to unfeasible or illogical solutions for the human.

In order to bridge this gap, we have proposed[4] to extend the classical representation of HATP by adding, on the one hand, the possibility to model a different knowledge for each agent and, on the other hand, the possibility to consider that an agent may or may not know some information.

2.10.1 The agents: myself and the others

In order to specify which agent is the robot or, more generally, the agent for which the system plans, we use the keyword `myself` instead of declaring a new agent. For example, if JIDO is the robot and HERAKLES is a human, the initialization will look like:

```
// step 3:creation of entities
JIDO = myself;
HERAKLES = new Agent;
```

2.10.2 Beliefs representation

To model different beliefs for the agents, the HATP formalism is extended using Multiple Values State Variables (MVSV). A multiple values state variable V is instantiated from a domain Dom and for each agent $a \in A$ the variable V has an instance $V(A) = v \in Dom$. For example:

```
// The agents have a different belief about the location of the tape:
WHITE_TAPE(JIDO).isIn = PINK_TRASHBIN;
WHITE_TAPE(HERAKLES).isIn = BLUE_TRASHBIN;
```

By default, in order to simplify and clarify the planning domain, only attributes for which the agents have a different belief, and only for agents different from the planning robot, are modeled with the MVSV formalism.

4 Guitton, Warnier and Alami - “Belief Management for HRI Planning” in proc. BNC@ECAI, 2012

```
// The agents have a different belief about the location of the tape:
BLACK_TAPE.isIn = PINK_TRASHBIN; // for all agents
WHITE_TAPE.isIn = PINK_TRASHBIN; // for Jido
WHITE_TAPE(HERAKLES).isIn = BLUE_TRASHBIN;
```

2.10.3 Known and unknown information

The agents' beliefs model includes the notion of *known* and *unknown* information. When an agent has no information about an entity attribute, the value associated to this property is set to unknown. When an agent, different from the the agent myself, knows an information (unknown for myself), this value is set to known.

```
// Herakles doesn't known where is the tape
WHITE_TAPE.isIn = PINK_TRASHBIN; // for Jido
WHITE_TAPE(HERAKLES).isIn = unknown;
```

An other example using known:

```
// Jido doen't know where is the tape, but Herakles knows
WHITE_TAPE.isIn = unknown; // for Jido
WHITE_TAPE(HERAKLES).isIn = known;
```

When an agent has no information about an entity, *i.e.*, all the attributes of this entity should be set to unknown, we simplify the representation by:

```
// Herakles has no information on the tape
WHITE_TAPE(HERAKLES) = unknown;
```

With this representation, we assume that even if the agent has no information concerning the object, it is informed about the existence of this object.

2.10.4 Communication actions

A communication action is a specific action that takes as parameters two agents, the emitter and the receiver, and a subject which is represented by an entity and an attribute. The prototype for a communication action is:

```
CommAction name (Agent A, Agent B, Entity E, Attribute T) {
    preconditions {...};
    effects {...};
    cost {...};
    duration {...}
}
```

The aim of a communication action is to transmit a value from an agent knowledge to another agent knowledge and corresponds to the effect:

```
E(B).T = E(A).T
```

This effect is implicit to this kind of action, *i.e.*, the domain designer does not need to specify it. But some extra effects can be added.

In order to fit the domain formalism, if the parameter corresponding to the attribute is set to `NULL`, then all the attributes of the concerned entity are transmitted from the agent A to the agent B. Otherwise, only the value of the specified attribute is communicated.

HATP with Belief Management distinguishes three types of communication: *information*, *contradiction* and *question*. The action of type information aims at giving an information when the attribute in unknown in the beliefs of the receiver. The planner produces an action of type contradiction when the attribute, for an agent is different from the attribute for myself. The question is used when a value is unknown for myself and known for another agent.

2.10.5 General communication method

In order to let planning domain designer name the communication action as he would do for classical actions, the communication actions are linked to the concepts of information, contradiction and question through a special method called `beliefManagement`:

```
beliefManagement {
    information { GiveInformationAbout; }
    contradiction { ForceInformation; }
    question { AskForInformation; }
}
```

with, for example, the communication action of type information:

```
commAction GiveInformationAbout(Agent A, Agent B, Entity E, Attribute T) {
    preconditions {};
    effects {
        FORALL(Agent C, {C != A;}, {E(C).T = E(A).T;}); // co-presence
    };
    cost{cost1()};
    duration{duration1()};
}
```

Each communication action must have been defined previously in the planning domain before the definition of the general communication method.

3 Installing HATP

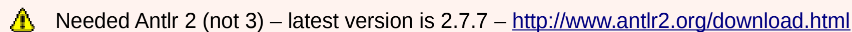
In order to be able to produce a plan, it is necessary to install the main planner module and the communication module. A graphical viewer is available to visualize plans. A C++ library is also available and aims at manipulating the output of HATP.

All these modules are available under GIT on the LAAS / RIA / RIS GIT server (trac) at `trac.laas.fr/git/robots/`

3.1 HATPOnboard

HATPOnboard contains the core of the planning system. It is available through GIT at the address `trac.laas.fr/git/robots/HATPOnboard` and relies on some dependencies (boost, antlr, MsgConnector)

```
// installing HATPOnboard
> git clone ssh://trac.laas.fr/git/robots/HATPOnboard
> cd HATPOnboard
> mkdir build ; cd build
> cmake .. ; make
```


Needed Antlr 2 (not 3) – latest version is 2.7.7 – http://www.antlr2.org/download.html

3.2 MsgConnector

MsgConnector is a package allowing the communication between HATPOnboard and other modules. It is composed of a server and a client API (C++) as well as some predefined bridges. It is available at `trac.laas.fr/git/robots/MsgConnector`. During the installation, it is possible to select which bridges should be compiled and installed (with ccmake)

```
// installing MsgConnector
> git clone ssh://trac.laas.fr/git/robots/MsgConnector
> cd MsgConnector
> mkdir build ; cd build
> cmake .. ; make ; make install
```

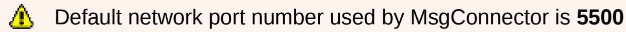
Available bridges are:

- MSGCONNECTOR-PRINT-BRIDGE: a simple text viewer
- MSGCONNECTOR-OPRS-BRIDGE: to communicate with SHARY (mp-oprs, boost, libHATP)
- MSGCONNECTOR-ORO-BRIDGE: to communicate with the ORO server (liboro, boost)
- MSGCONNECTOR-YARP-BRIDGE: to communicate with a YARP module (YARP libs)

By default, MsgConnector is compiled without any bridge. To select bridges to compile, set bridge options to ON with:

```
> ccmake ..
```

MsgConnector contains also a small module called `HATPGoalTester` allowing to send a goal to HATPOnboard like a supervisor would do it. It can be found in the tools folder.

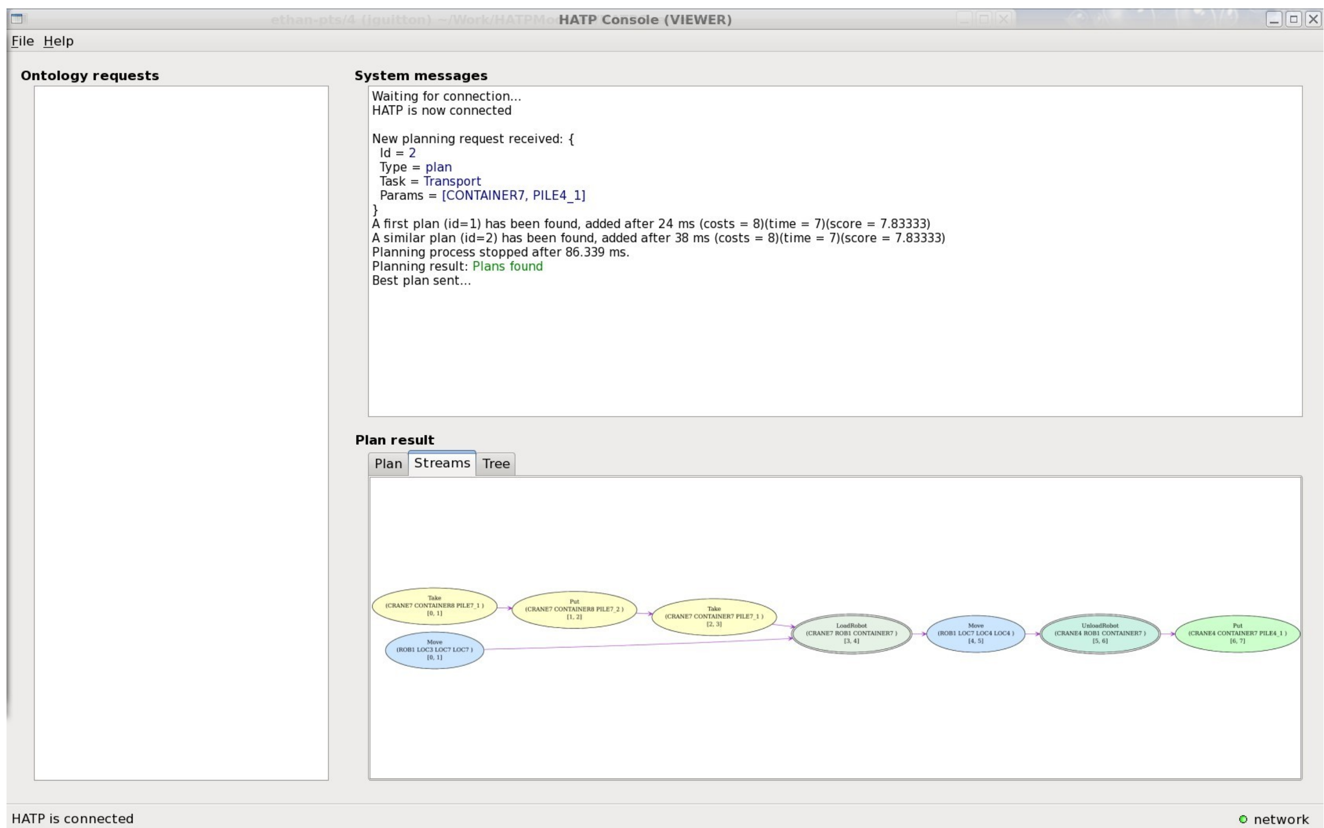

Default network port number used by MsgConnector is **5500**

3.3 HATPConsole

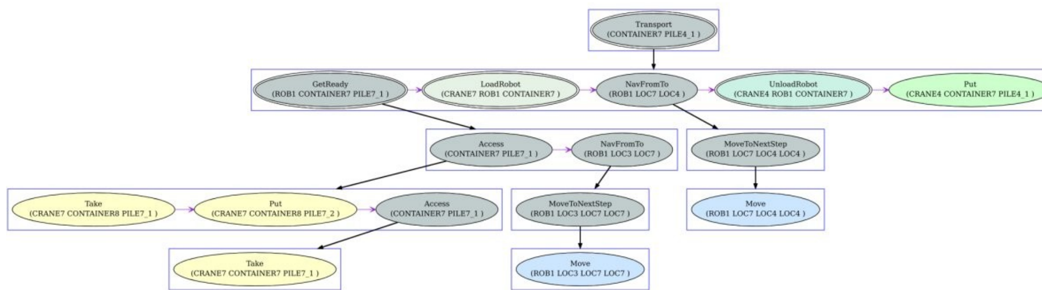
HATPConsole is a graphical viewer for the plans produced by HATPOnboard. This module relies on MsgConnector, boost, Qt and GraphViz (libGVC). It is available at the address `trac.laas.fr/git/robots/HATPConsole`.

```
// installing HATPConsole
> git clone ssh://trac.laas.fr/git/robots/HATPConsole
> cd HATPConsole
> mkdir build ; cd build
> cmake .. ; make
```

The console looks like (in standalone mode / without ontology requests):



It is possible to visualize the plan under a text form, under a set of streams or to visualize the corresponding planning tree:



3.4 LibHATP

LibHATP is a C++ library allowing to manipulate easily plans produced by HATPOnboard. It is available at `trac.laas.fr/git/robots/LibHATP` and depends only on boost regex.

```
// installing LibHATP
> git clone ssh://trac.laas.fr/git/robots/LibHATP
> cd LibHATP
> mkdir build ; cd build
> cmake .. ; make ; make install
```

LibHATP contains a doxygen documentation available after typing:

```
> make doc
```

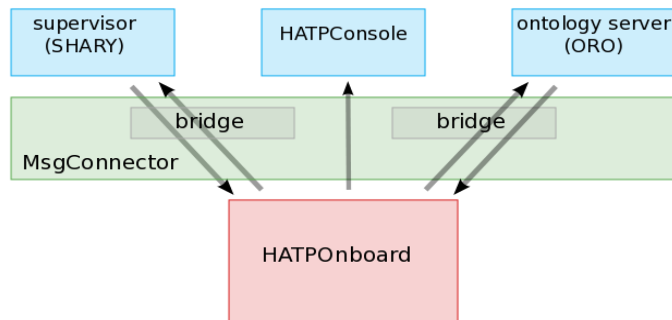
4 Running HATP

In this section, we give a brief overview of how works HATP and how to launch the planning system.

4.1 How works HATP ?

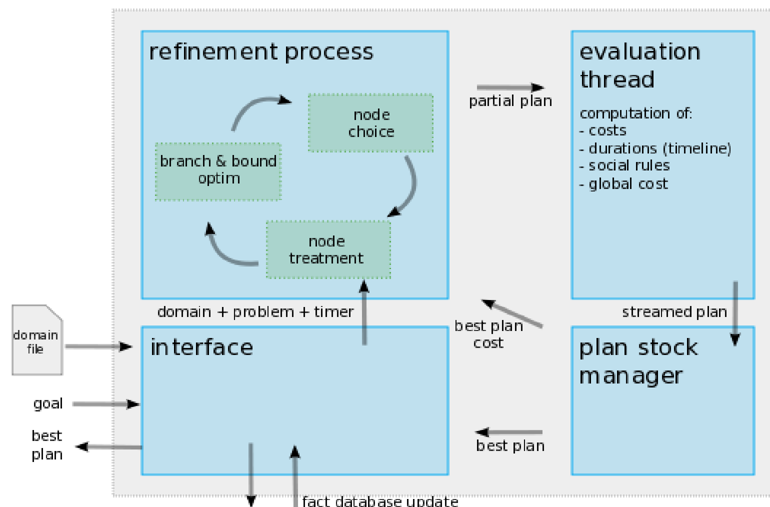
The entire planning system is composed of a main part: the planning process, and a set of other modules allowing to communicate, visualize plans and manipulate them.

4.1.1 Overview of the planning system



The supervisor sends to HATPOnboard a planning request. HATPOnboard gets the initial state by sending ontology requests to ORO then computes possible plans. It sends back to the supervisor the best plan found.

4.1.2 Planning process



The HTN planning process is located in the source file `hatpPlanningSession.cc` in the `planning/` folder. This process is composed of a main loop that refines the planning tree by decomposing tasks into sub-tasks (classical HTN process). Then, when a plan is found, the system transforms it into streams, computes the global cost (`plan.cc`) and stocks it (`planStockManager.cc`).

4.1.3 Important files of HATPOnboard

Most of changes that can be made to HATPOnboard as, for example, changing the network interface, adding a new functionality, will take place in one of the following files:

C++ files	
`onboard/mainOnboardHATP.cc`	Main file of HATPOnboard
`onboard/requestLexer.hh`	Message request parser
`parsing/HATP.g`	Domain parser (antlr language)
`planning/hatpPlanningSession.cc`	Main loop of the planning process
`plan/plan.cc`	Internal structure of a plan
`plan/TimeProjection.cc`	Parallelization process

4.2 Launching the system

The first module that needs to be launch is MsgConnector. This module and the others can be executed through a script present in the current folder.

```
# in MsgConnector:
> build/server/MsgServer
#or
> ./run
```

Then, HATPOnboard, HATPConsole and bridges can be launched. For HATPOnboard, the domain must be parsed in order to create corresponding C++ code, then recompiled:

```
# in HATPOnboard:
> build/parser/HATPparser DOMAIN_FILE COST_FILE
> cd build ; make clean ; make
#or
> ./parse (after having configured it)
```

After parsing the domain, HATPOnboard can be launched.

```
# in HATPOnboard:
> build/onboard/HATPonboard MP_SERVER ORO_NAME
#or
> ./run
```

For standalone version (without connection to ORO or other knowledge-based server):

```
# in HATPOnboard:
> build/onboard/HATPonboard MP_SERVER
#or
> ./sa_run
```

4.3 Communication with HATP

The communication can be seen at two levels: A higher level or semantic level describing in which formalism to exchange data and a lower level or network level that manages the transfers of information.

For the lower level, MsgConnector offers an API allowing to easily send and receive messages from an external module.

To communicate and exchange data, the HATP planning system uses messages based on the JSON[5] syntax.

4.3.1 Communication API

The requests are received and sent by HATP through the MsgConnector server. MsgConnector proposes a C++ library to write your own network client. The prototypes of available methods are given:

```
// msgClient.hh
// constructor and destructor
msgClient();
~msgClient();
// connect and disconnect
bool connect(std::string name, std::string server, unsigned int port);
void disconnect();
// test if connected
bool isConnected();
// test if a message has been received
bool isMessageWaiting();
// send a message
bool sendMessage(std::string dest, std::string content);
// get a message (non-blocking and blocking method)
std::pair<std::string, std::string> getMessage();            // <from, message>
std::pair<std::string, std::string> getBlockingMessage();    // <from, message>
```

Here is an example of such a client:

```
// a simple module that sends goals and receive plan
string myFirstPlanRequest("...");
msgClient* mp = new msgClient();
mp->connect("GOALTESTER", "localhost", 5500);
if(mp->isConnected()) {
    mp->sendMessage("HATP", myFirstPlanRequest);
    pair<string, string> result = mp->getBlockingMessage();
    cout << "plan: " << result.second;
}
```

5 JavaScript Object Notation - http://www.json.org

4.3.2 Basic communication syntax

All the incoming requests must begin with:

```
{"HATP-REQ": {
    "REQ-TYPE":
```

The planner main module is able to handle some requests:

Request types	
plan	Ask HATP to compute plans and send back the best one
replan	Ask HATP to replan in case of failure (not implemented)
getTasks	Ask HATP the list of high-level tasks
getActions	Ask HATP the list of primitive actions
setMaxTime	Set the planning time limit (in seconds)

A HATP answer begins with the following syntax:

```
{"HATP-REP": {
```

4.3.3 Requests syntax

A planning request contains some other fields identifying the task to plan. The complete planning request syntax is:

```
{"HATP-REQ": {
    "REQ-TYPE": "plan",
    "REQ-ID": NUMBER,                   // an id referring to this planning request
    "TASK-NAME": STRING,                // the goal-task name
    "PARAMETERS": ARRAY[STRING]         // list of parameters for this goal
}}
```

For example:

```
{"HATP-REQ": {
    "REQ-TYPE": "plan",
    "REQ-ID": 1,
    "TASK-NAME": "Transport",
    "PARAMETERS": ["CONTAINER7", "PILE4_1"]
}}
```

To get the list of tasks or actions, only the `REQ-TYPE` field is important:

```
{"HATP-REQ": {
    "REQ-TYPE": "getTasks"          // or getActions
}}
```

To set the planning time limit. The value is the number of seconds:

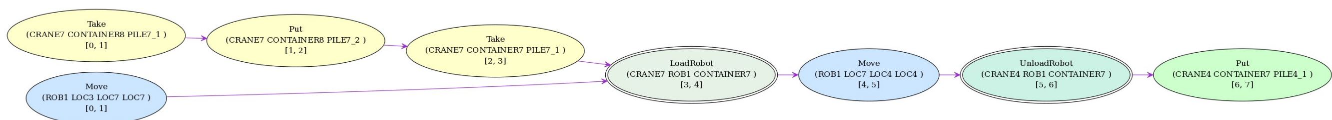
```
{"HATP-REQ": {
    "REQ-TYPE": "setMaxTime",
    "MAXTIME": NUMBER
}}
```

4.3.4 Answer of HATP

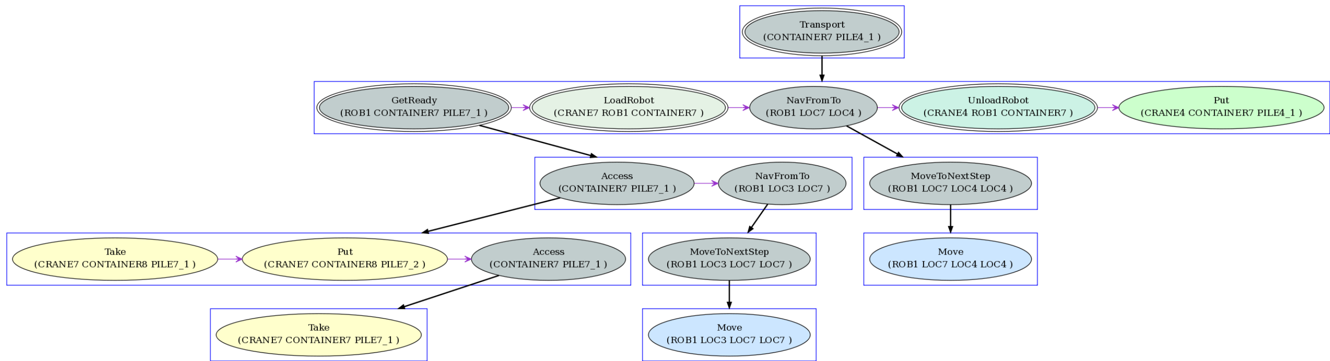
After having computed all the possible plans or reached the planning time limit, HATP sends back the best plan found using the syntax:

```
{"HATP-REP": {
    "REQ-ID": NUMBER,                         // corresponds to the id of the planning request
    "REPORT": STRING,                         // result of the planning phase
    "NODES": ARRAY[                           // list of nodes of the resulting plan
        {"NODE": {
            "NODE-ID": NUMBER,                // id of the action
            "NODE-TYPE": STRING,              // type of the node: metatask or action
            "ACTION-NAME": STRING,            // name of the task
            "AGENTS": ARRAY[STRING],          // list of agent involved in the action
            "START-TIME": NUMBER,             // start time (second) of the action
            "END-TIME": NUMBER,               // end time (second) of the action
            "PARAMETERS": ARRAY[STRING]       // ordered unified parameters of the action
        }}
    ],
    "STREAMS": {                              // sub-structure describing the set of streams
        "ROOTS": ARRAY[NUMBER],               // root node of each stream
        "LINKS": ARRAY[                       // list of causal links between nodes
            {"LINK": {
                "TYPE": STRING,               // should be: causal
                "FROM": NUMBER,               // id of the predecessor action
                "TO": NUMBER                  // id of the successor action
            }}
        ]
    },
    "TREE": {                                 //  sub-structure describing the planning tree
        "ROOTS": ARRAY[NUMBER],               // root node(s) of the planning tree
        "LINKS": ARRAY[                       // list of links between nodes
            {"LINK": {
                "TYPE": STRING,               // can be causal or hierarchical
                "FROM": NUMBER,               // id of the predecessor action
                "TO": NUMBER                  // id of the successor action
            }}
        ]
    }
}}
```

This result message contains all the necessary information to exploit the plan produced by HATP. For example, HATPConsole, when receiving this kind of message, is able to construct the following graphs:



streams extracted from the answer message



Tree extracted from the answer message

For answers to other requests (`getTasks` and `getActions`) the syntax is:

```
{"HATP-REP": {
    "RESULT": ARRAY[ {
        "TASK-NAME": STRING,
        "PARAMETERS": ARRAY[STRING]
    }]
}}
```

4.4 Manipulation of plans

It is not mandatory to manipulate directly the previous syntax to extract data from a plan result message sent by HATPOnboard. You can use the plan manipulation C++ library called LibHATP.

4.4.1 LibHATP

In order to manipulate automatically the plan received by HATP, the message needs to be parsed. LibHATP contains such a parser and transforms the plan result message into structures that can be manipulated directly in C++.

LibHATP available C++ classes are :

C++ classes	
`hatpPlan`	Main class representing a plan
`hatpStreams`	Representation of the streams extracted from the plan message
`hatpTree`	Representation of the tree extracted from the plan message
`hatpNode`	Class representing a planning node
`hatpTaskList`	Sequence of nodes

To manipulate a plan received from HATPOnboard, you just need to create a `hatpPlan` with the content of the message in argument. The parsing will be done automatically:

```
hatpPlan* myPlan = new hatpPlan(removeFormatting(result.second));
```

Each class contains all the necessary accessors to manipulated the plan data. For example, for the `hatpNode` class, the public methods are: `getID()`, `getType()`, `getName()`, `getAgents()`, `getStartTime()`, `getEndTime()`, `getParameters()`, `getRootNode()`, `getSubNodes()`, `getTreePredecessors()`, `getTreeSuccessors()`, `getStreamPredecessors`, `getStreamSuccessors()`, `isAction()`, `toString()`, ...

For more details, please refer to the doxygen documentation of this API.

4.4.2 A simple application

For example, we can extend our previous client which sends a goal to HATP and receives the resulting plan, like this:

```
// a simple module that counts actions per agent and give the duration
// for the DWR domain

#include "msgClient.hh"
#include "hatpPlan.hh"

string myFirstPlanRequest("{\"HATP-REQ\":{\"REQ-TYPE\": \"plan\", \"REQ-ID\": 1,
        \"TASK-NAME\": \"Transport\", \"PARAMETERS\": [\"CONTAINER7\", \"PILE4_1\"]}}");
msgClient* mp = new msgClient();
mp->connect("GOALTESTER", "localhost", 5500);
if(mp->isConnected()) {
    mp->sendMessage("HATP", myFirstPlanRequest);
    pair<string, string> result = mp->getBlockingMessage();
    // we get the plan
    string toAnalyse(result.second);
    removeFormatting(toAnalyse);
    hatpPlan* myPlan = new hatpPlan(result.second);
    if(myPlan->isPlanValid()) {
        cout << "We have received a plan! (id = "<< myPlan->getID() << ")" << endl
        vector<string> agents = myPlan->getAgentsList();
        for(unsigned int i = 0 ; i < agents.size() ; i++) {
            unsigned int endTime = 0;
            vector<HATPNode*> nodes = myPlan->getStreams()->getNodesForAgent(agents[i]);
            cout << agents[i] << " has participated to " << nodes.size() << " actions." << endl;
            for(unsigned int j = 0 ; j < nodes.size() ; j++) {
                if(endTime < nodes[j]->getEndTime()) endTime = nodes[j]->getEndTime();
            }
            cout << "He has finished its work after " << endTime << " seconds." << endl;
        }
    }
}
mp->disconnect();
```

Is is mandatory to call the function `removeFormatting(string)` before creating a plan structure!

Output of this small example should look like:

```
We have received a plan! (id = 1)
CRANE7 has participated to 4 actions.
He has finished its work after 4 seconds.
ROB1 has participated to 4 actions.
He has finished its work after 6 seconds.
CRANE4 has participated to 2 actions.
He has finished its work after 7 seconds.
```

- That's all folks! -

A The “Clean The Table” domain

```
factdatabase {

    // entity types
    define entityType Table;
    define entityType Trashbin;
    define entityType GameArtifact;

    // entity attributes
    define entityAttributes Agent {
        static atom string type;
        dynamic atom GameArtifact hasInRightHand;
    }
    define entityAttributes Table {
        static atom string hasColor;
    }
    define entityAttributes Trashbin {
        static atom string hasColor;
        dynamic set Agent isReachable;
    }
    define entityAttributes GameArtifact {
        static atom string hasColor;
        dynamic set Agent isVisible;
        dynamic set Agent isReachable;
        dynamic atom Table isOn;
        dynamic atom Trashbin isIn;
    }

    // entity declaration
    ACHILE_HUMAN1 = new Agent;
    JIDOKUKA_ROBOT = new Agent;
    HRP2TABLE = new Table;
    BLUE_TRASHBIN = new Trashbin;
    PINK_TRASHBIN = new Trashbin;
    BLACK_TAPE = new GameArtifact;
    GREY_TAPE = new GameArtifact;
    BLUE_CUBE = new GameArtifact;
    //BLUE_CUBE_2 = new GameArtifact;
    RED_CUBE = new GameArtifact;
    //RED_CUBE_2 = new GameArtifact;

    // static initialisation
    ACHILE_HUMAN1.type = "Human";
    JIDOKUKA_ROBOT.type = "Robot";
    BLUE_TRASHBIN.hasColor = "blue";
    PINK_TRASHBIN.hasColor = "pink";
    BLACK_TAPE.hasColor = "black";
    GREY_TAPE.hasColor = "grey";
    BLUE_CUBE.hasColor = "blue";
    //BLUE_CUBE_2.hasColor = "blue";
    RED_CUBE.hasColor = "red";
    //RED_CUBE_2.hasColor = "red";


}

// ----------------------------------------------

HTN {

    action TakeObject(Agent A, GameArtifact C, Table T) {
        preconditions {
            C.isOn == T;
            A >> C.isVisible;
            A >> C.isReachable;
            A.hasInRightHand == NULL;
        };
        effects {
            FORALL(Agent Ag, {Ag >> C.isVisible;}, {C.isVisible =>> Ag;});
            A.hasInRightHand = C;
            C.isOn = NULL;
        };
        cost{cost1()};
        duration{duration1()};
    }
```

```
action PutObject(Agent A, GameArtifact C, Table T) {
    preconditions {
        A.hasInRightHand == C;
    };
    effects {
        A.hasInRightHand = NULL;
        C.isOn = T;
        FORALL(Agent Ag1, {}, {C.isVisible <<= Ag1;});
        FORALL(Agent Ag2, {}, {C.isReachable <<= Ag2;});
    };
    cost{cost1()};
    duration{duration1()};
}

action PutObjectUnvisible(Agent A, GameArtifact C, Table T) {
    preconditions {
        A.hasInRightHand == C;
    };
    effects {
        A.hasInRightHand = NULL;
        C.isOn = T;
        C.isVisible <<= A;
        C.isReachable <<= A;
        FORALL(Agent Ag1, {Ag1 != A;}, {C.isVisible =>> Ag1;});
        FORALL(Agent Ag2, {Ag2 != A;}, {C.isReachable =>> Ag2;});
    };
    cost{cost1()};
    duration{duration1()};
}

action PutObjectVisible(Agent A, GameArtifact C, Table T) {
    preconditions {
        A.hasInRightHand == C;
    };
    effects {
        A.hasInRightHand = NULL;
        C.isOn = T;
        C.isReachable <<= A;
        FORALL(Agent Ag, {}, {C.isVisible <<= Ag;});
    };
    cost{cost1()};
    duration{duration1()};
}

action PutObjectVisibleAndReachable(Agent A, GameArtifact C, Table T) {
    preconditions {
        A.hasInRightHand == C;
    };
    effects {
        A.hasInRightHand = NULL;
        C.isOn = T;
        C.isVisible <<= A;
        FORALL(Agent Ag1, {}, {C.isVisible <<= Ag1;});
        FORALL(Agent Ag2, {}, {C.isReachable <<= Ag2;});
    };
    cost{cost1()};
    duration{duration1()};
}

action ThrowObject(Agent A, GameArtifact C, Trashbin T) {
    preconditions {
            A.hasInRightHand == C;
            A >> T.isReachable;
    };
    effects {
        A.hasInRightHand = NULL;
        C.isIn = T;
    };
    cost{cost1()};
    duration{duration1()};
}
```

```
action GiveObject(Agent A1, GameArtifact C, Agent A2) {
    preconditions {
        A1.hasInRightHand == C;
        A2.hasInRightHand == NULL;
    };
    effects {
        A1.hasInRightHand = NULL;
        A2.hasInRightHand = C;
    };
    cost{cost10()};
    duration{duration1()};
}

// ---------------------------------------------

method Get(Agent A, GameArtifact C) {
    empty{A.hasInRightHand == C;};
    {
        preconditions{
            A.hasInRightHand == NULL;
            A >> C.isReachable;
        };
        subtasks {
            T = SELECT(Table, {C.isOn == T;});
            1:TakeObject(A, C, T);
        };
    }
    {
        preconditions{
            A.hasInRightHand != NULL;
            A >> C.isReachable;
        };
        subtasks {
            C2 = SELECT(GameArtifact, {A.hasInRightHand == C2;});
            T = SELECT(Table, {C.isOn == T;});
            1:Place(A, C2, T);
            2:TakeObject(A, C, T)>1;
        };
    }
    {
        preconditions{
            A.hasInRightHand == NULL;
            A !>> C.isReachable;
        };
        subtasks {
            B = SELECT(Agent, {B >> C.isReachable;});
            1:Get(B, C);
            2:Give(B, C, A)>1;
        };
    }
    {
        preconditions{
            A.hasInRightHand != NULL;
            A !>> C.isReachable;
        };
        subtasks {
            C2 = SELECT(GameArtifact, {A.hasInRightHand == C2;});
            B = SELECT(Agent, {B >> C.isReachable;});
            T = SELECT(Table, {});
            1:Place(A, C2, T);
            2:Get(B, C);
            3:Give(B, C, A)>1,>2;
        };
    }
}

method Place(Agent A, GameArtifact C, Table T) {
    empty{C.isOn == T;};
    {
        preconditions {};
        subtasks {
            1:Get(A, C);
            2:PutObject(A, C, T)>1;
        };
    }
}
```

```
    method Give(Agent A, GameArtifact C, Agent B) {
        empty{B.hasInRightHand == C;};
        {
            preconditions {B.hasInRightHand == NULL;};
            subtasks {
                1:Get(A, C);
                2:GiveObject(A, C, B)>1;
            };
        }
        {
            preconditions {B.hasInRightHand == NULL;};
            subtasks {
                T = SELECT(Table, {});
                1:Get(A, C);
                2:PutObjectVisibleAndReachable(A, C, T)>1;
                3:Get(B, C)>2;
            };
        }
    }

    method Hide(Agent A, GameArtifact C, Agent B) {
        empty{A.hasInRightHand != C; B >> C.isVisible;};
        {
            preconditions {};
            subtasks {
                T = SELECT(Table, {});
                1:Get(A, C);
                2:PutObjectUnvisible(A, C, T)>1;
            };
        }
    }

}

// ---------------------------------------------

timePart { priority = 0; }
```

B The “Dock Worker Robot” domain

This domain has been translated from SHOP2 formalism.

```
factdatabase {

    // entity types
    define entityType Location;
    define entityType Path;
    define entityType Pile;
    define entityType Crane;
    define entityType Container;

    // entity attributes
    define entityAttributes Agent {
        static atom string type; // ROBOT or CRANE
        dynamic atom Location at; // also used for belong
        dynamic atom Container loaded; // also used for holding, unloaded, empty
        dynamic set Location path; // for list path

    }
    define entityAttributes Location {
        static set Location adjacent;
        static atom number xcoord;
        static atom number ycoord;
        dynamic atom bool occupied;
        dynamic atom Agent isForbiddenBy; // for list forbid

    }
    define entityAttributes Pile {
        static atom Location attached;
        dynamic set Container contains;
    }
    define entityAttributes Container {
        dynamic atom Pile in;
        dynamic atom Pile top;
        dynamic atom Container on;
    }

// ------------------------------------------------
// INITIAL STATE

    // entity declaration
    ROB1 = new Agent; ROB2 = new Agent; ROB3 = new Agent;
    CRANE1 = new Agent; CRANE2 = new Agent; CRANE3 = new Agent; CRANE4 = new Agent; CRANE5 = new Agent;
    CRANE6 = new Agent; CRANE7 = new Agent; CRANE8 = new Agent; CRANE9 = new Agent; CRANE10 = new Agent;
    LOC1 = new Location; LOC2 = new Location; LOC3 = new Location; LOC4 = new Location; LOC5 = new Location;
    LOC6 = new Location; LOC7 = new Location; LOC8 = new Location; LOC9 = new Location; LOC10 = new Location;
    PILE1_1 = new Pile; PILE1_2 = new Pile; PILE2_1 = new Pile; PILE2_2 = new Pile;
    PILE3_1 = new Pile; PILE3_2 = new Pile; PILE4_1 = new Pile; PILE4_2 = new Pile;
    PILE5_1 = new Pile; PILE5_2 = new Pile; PILE6_1 = new Pile; PILE6_2 = new Pile;
    PILE7_1 = new Pile; PILE7_2 = new Pile; PILE8_1 = new Pile; PILE8_2 = new Pile;
    PILE9_1 = new Pile; PILE9_2 = new Pile; PILE10_1 = new Pile; PILE10_2 = new Pile;
    CONTAINER1 = new Container; CONTAINER2 = new Container; CONTAINER3 = new Container; CONTAINER4 = new Container;
    CONTAINER5 = new Container; CONTAINER6 = new Container; CONTAINER7 = new Container; CONTAINER8 = new Container;
    CONTAINER9 = new Container; CONTAINER10 = new Container;

    // static initialisation
    ROB1.type = "ROBOT"; ROB2.type = "ROBOT"; ROB3.type = "ROBOT";
    CRANE1.type = "CRANE"; CRANE2.type = "CRANE"; CRANE3.type = "CRANE"; CRANE4.type = "CRANE";
    CRANE5.type = "CRANE"; CRANE6.type = "CRANE"; CRANE7.type = "CRANE"; CRANE8.type = "CRANE";
    CRANE9.type = "CRANE"; CRANE10.type = "CRANE";
    PILE1_1.attached = LOC1; PILE1_2.attached = LOC1; PILE2_1.attached = LOC2; PILE2_2.attached = LOC2;
    PILE3_1.attached = LOC3; PILE3_2.attached = LOC3; PILE4_1.attached = LOC4; PILE4_2.attached = LOC4;
    PILE5_1.attached = LOC5; PILE5_2.attached = LOC5; PILE6_1.attached = LOC6; PILE6_2.attached = LOC6;
    PILE7_1.attached = LOC7; PILE7_2.attached = LOC7; PILE8_1.attached = LOC8; PILE8_2.attached = LOC8;
    PILE9_1.attached = LOC9; PILE8_2.attached = LOC8; PILE10_1.attached = LOC10; PILE10_2.attached = LOC10;
    CRANE1.at = LOC1; CRANE2.at = LOC2; CRANE3.at = LOC3; CRANE4.at = LOC4; CRANE5.at = LOC5;
    CRANE6.at = LOC6; CRANE7.at = LOC7; CRANE8.at = LOC8; CRANE9.at = LOC9; CRANE10.at = LOC10;
    LOC1.adjacent <<= LOC5;
    LOC2.adjacent <<= LOC5;
    LOC3.adjacent <<= LOC7;
    LOC4.adjacent <<= LOC7;
    LOC5.adjacent <<= LOC1; LOC5.adjacent <<= LOC2; LOC5.adjacent <<= LOC6;  LOC5.adjacent <<= LOC9;
    LOC6.adjacent <<= LOC5; LOC6.adjacent <<= LOC7;
    LOC7.adjacent <<= LOC3; LOC7.adjacent <<= LOC4; LOC7.adjacent <<= LOC6; LOC7.adjacent <<= LOC8;
    LOC8.adjacent <<= LOC7;
```

```
    LOC9.adjacent <<= LOC5; LOC9.adjacent <<= LOC10;
    LOC10.adjacent <<= LOC9;

    // default init
    LOC1.occupied = false; LOC2.occupied = false; LOC3.occupied = false; LOC4.occupied = false;
    LOC5.occupied = false; LOC6.occupied = false; LOC7.occupied = false; LOC8.occupied = false;
    LOC9.occupied = false; LOC10.occupied = false;

    // dynamic initialisation (initial state)
    ROB1.at = LOC3; LOC3.occupied = true;
    ROB2.at = LOC6; LOC6.occupied = true;
    ROB3.at = LOC10; LOC10.occupied = true;
    CONTAINER1.in = PILE1_1; PILE1_1.contains <<= CONTAINER1;
    CONTAINER2.in = PILE1_1; CONTAINER2.on = CONTAINER1; PILE1_1.contains <<= CONTAINER1;
    CONTAINER3.top = PILE1_1; CONTAINER3.in = PILE1_1; CONTAINER3.on = CONTAINER2; PILE1_1.contains <<= CONTAINER3;
    CONTAINER4.in = PILE1_2; PILE1_2.contains <<= CONTAINER4;
    CONTAINER5.in = PILE1_2; CONTAINER5.on = CONTAINER4; PILE1_2.contains <<= CONTAINER5;
    CONTAINER6.top = PILE1_2; CONTAINER6.in = PILE1_2; CONTAINER6.on = CONTAINER5; PILE1_2.contains <<= CONTAINER6;
    CONTAINER7.in = PILE7_1; PILE7_1.contains <<= CONTAINER7;
    CONTAINER8.top = PILE7_1; CONTAINER8.in = PILE7_1; CONTAINER8.on = CONTAINER7; PILE7_1.contains <<= CONTAINER8;
    CONTAINER9.in = PILE10_1; PILE10_1.contains <<= CONTAINER9;
    CONTAINER10.top = PILE10_1; CONTAINER10.in = PILE10_1; CONTAINER10.on = CONTAINER9;
    PILE10_1.contains <<= CONTAINER10;

}

// ---------------------------------------------

HTN {

    // moves a robot between two adjacent locations
    action Move(Agent R, Location From, Location To, Location FinalDest) {
        preconditions {
            R.type == "ROBOT";
            To >> From.adjacent;
            R.at == From;
            To.occupied == false;
        };
        effects {
            R.at = To;
            From.occupied = false;
            To.occupied = true;
            R.path <<= To; // added
            IF{From !>> R.path;}{R.path <<= From;} // added
            IF{To.isForbiddenBy == R;}{To.isForbiddenBy = NULL;}
            IF{To == FinalDest;}{ // then clean
                FORALL(Location LocP, {LocP >> R.path;}, {R.path =>> LocP;});
            }
        };
        cost{costFn(1)};
        duration{durationFn(1, 1)};
    }

    // loads an empty robot with a container held by a crane in the same location
    action LoadRobot(Agent G, Agent R, Container C) {
        preconditions {
            R.type == "ROBOT";
            G.type == "CRANE";
            R.at == G.at;
            G.loaded == C;
            R.loaded == NULL;
        };
        effects {
            G.loaded = NULL;
            R.loaded = C;
        };
        cost{costFn(1)};
        duration{durationFn(1, 1)};
    }

    // unloads a robot hloding a container with a nearby crane
    action UnloadRobot(Agent G, Agent R, Container C) {
        preconditions {
            R.type == "ROBOT";
            G.type == "CRANE";
            R.at == G.at;
            R.loaded == C;
            G.loaded == NULL;
        };
        effects {
```

```
            R.loaded = NULL;
            G.loaded = C;
        };
        cost{costFn(1)};
        duration{durationFn(1, 1)};
    }

    // takes a container from the top of a pile with a crane in same location
    action Take(Agent R, Container C, Pile P) {
        preconditions {
            R.type == "CRANE";
            R.at == P.attached;
            R.loaded == NULL;
            C.top == P;
        };
        effects {
            R.loaded = C;
            FORALL(Container C2, {C.on == C2;}, {C2.top = P;});
            C.top = NULL;
            C.in = NULL;
            C.on = NULL;
            P.contains =>> C;
        };
        cost{costFn(1)};
        duration{durationFn(1, 1)};
    }

    // puts a container helds by a crane on the top of a pile in same location
    action Put(Agent R, Container C, Pile P) {
        preconditions {
            R.type == "CRANE";
            R.at == P.attached;
            R.loaded == C;
        };
        effects {
            R.loaded = NULL;
            FORALL(Container C2, {C2.top == P;}, {C2.top = NULL; C.on = C2;});
            C.top = P;
            C.in = P;
            P.contains <<= C;
        };
        cost{costFn(1)};
        duration{durationFn(1, 1)};
    }

    // set a location as forbidden for backtracking (new action added to add a loc to ?forbid)
    action AddForbidden(Agent R, Agent R2, Location L) {
        preconditions {};
        effects {
            L.isForbiddenBy = R;
        };
        cost{costFn(0)};
        duration{durationFn(0, 0)};
    }

    // Top-level task for transporting a container K to a pile Target
    method Transport(Container K, Pile Target) {
        // do nothing (done)
        empty{K.in == Target;};
        // container already in location
        {
            preconditions {
                EXIST(Pile Source1, {K.in == Source1;}, {Source1.attached == Target.attached;});
            };
            subtasks {
                R = SELECT(Agent, {R.type == "CRANE"; R.at == Target.attached;});
                S = SELECT(Pile, {K.in == S;});
                1:Access(K, S);
                2:Put(R, K, Target)>1;
            };
        }
        // else
        {
            preconditions {
                EXIST(Pile Source2, {K.in == Source2;}, {Source2.attached != Target.attached;});
            };
            subtasks {
                S = SELECT(Pile, {K.in == S;});
                R = SELECTORDERED(Agent, {R.type == "ROBOT";}, distanceMin(R.at, S.attached), <);
                G1 = SELECT(Agent, {G1.type == "CRANE"; G1.at == S.attached;});
```

```
            G2 = SELECT(Agent, {G2.type == "CRANE"; G2.at == Target.attached;});
            1: GetReady(R, K, S);
            2: LoadRobot(G1, R, K)>1;
            3: NavFromTo(R, S.attached, Target.attached)>2;
            4: UnloadRobot(G2, R, K)>3;
            5: Put(G2, K, Target)>4;
        };

    }
}

// Brings robot R to loc To and/or accesses the container K in pile P
method GetReady(Agent R, Container K, Pile P) {
    // not used
    empty {R.type == "TOTO";};
    // get container only
    {
        preconditions {
            R.at == P.attached;
        };
        subtasks {
            1: Access(K, P);
        };
    }
    // do both: get container + brings robot
    {
        preconditions {
            R.at != P.attached;
        };
        subtasks {
            From = SELECT(Location, {R.at == From;});
            To = SELECT(Location, {P.attached == To;});
            1: Access(K, P);
            2: NavFromTo(R, From, To)>1;
        };
    }
}

// Removes what's on top of container k to take it with crane C
method Access(Container K, Pile P) {
    // never used (so defined by an impossible cond)
    empty{P.attached == NULL;};
    // on top
    {
        preconditions {
            K.top == P;
        };
        subtasks {
            C = SELECT(Agent, {C.type == "CRANE"; C.at == P.attached;});
            1:Take(C, K, P);
        };
    }
    // else (K not on top but in, exist P2)
    {
        preconditions {
            K.top == NULL;
            K.in == P;
        };
        subtasks {
            C = SELECT(Agent, {C.type == "CRANE"; C.at == P.attached;});
            Ktop = SELECT(Container, {Ktop.top == P;});
            P2 = SELECT(Pile, {P2 != P; P2.attached == P.attached;});
            1:Take(C, Ktop, P);
            2:Put (C, Ktop, P2)>1;
            3:Access(K, P)>2;

        };
    }
}

// Top-level task to navigate a robot R To, search for a path...
method Navigate(Agent R, Location To) {
    empty {R.at == To;};
    {
        preconditions {};
        subtasks {
            From = SELECT(Location, {R.at == From;});
            1:NavFromTo(R, From, To);
        };
    }
```

```
}

// Navigation method avoiding loops
method NavFromTo(Agent R, Location From, Location To) {
    // do nothing - done
    empty {R.at == To;};
    // Adjacent destination
    {
        preconditions {
            To >> From.adjacent;
        };
        subtasks {
            1: MoveToNextStep(R, From, To, To);
        };
    }
    // else
    {
        preconditions {
            To !>> From.adjacent;
        };
        subtasks {
            Next = SELECTORDERED(Location, {Next >> From.adjacent; Next !>> R.path;}, distanceMin(Next, To), <);
            1: MoveToNextStep(R, From, Next, To);
            2: NavFromTo(R, Next, To)>1;
        };
    }
}

// one step in navigation ; freeing the next location if needed
method MoveToNextStep(Agent R, Location From, Location Next, Location To) {
    // never used
    empty{From == Next;};
    // adjacent node free
    {
        preconditions {
            Next.occupied == false;
        };
        subtasks {
            1: Move(R, From, Next, To);
        };
    }
    // adjacent node not free
    {
        preconditions {
            Next.occupied == true;
            EXIST(Agent R2, {R2.type == "ROBOT";}, {R2.at == Next;});
        };
        subtasks {
            R2 = SELECT(Agent, {R2.type == "ROBOT"; R2.at == Next;});
            1: FreeLocation(R, R2, From, Next, To);
            2: Move(R, From, Next, To)>1;
        };
    }
}

// Moves the R2 out of the way
method FreeLocation(Agent R, Agent R2, Location From, Location Next, Location To) {
    // never used
    empty{R == R2;};
    // there is free space
    {
        preconditions {
            EXIST(Location Lfree1, {Lfree1 >> Next.adjacent;},
                                   {Lfree1.occupied == false; Lfree1.isForbiddenBy == NULL;});
        };
        subtasks {
            L = SELECTORDERED(Location, {L >> Next.adjacent; L.occupied == false;L.isForbiddenBy == NULL;},
                                    distanceMin(L, To), >);
            1: Move(R2, Next, L, L);
        };
    }
    // Robot R can backtrack
    {
        preconditions {
            EXIST(Location Lfree3, {Lfree3 >> From.adjacent;}, {Lfree3.occupied == false;});
        };
        subtasks {
            LL = SELECT(Location, {LL >> From.adjacent; LL.occupied == false;});
            1: AddForbidden(R, R2, Next); // here next must be added to ?forbid
            2: Move(R, From, LL, To)>1;
```

```
                3: Move(R2, Next, From, From)>2;
                4: FreeLocation(R, R2, LL, From, To)>3;
                5: Move(R, LL, From, To)>4;
            };
        }
    }

    // Top-level task to transport all containers from SourcePile to TargetPile
    method TransportAll(Pile SourcePile, Pile TargetPile) {
        // never used
        empty {SourcePile == TargetPile;};
        // piles in same location
        {
            preconditions {
                SourcePile.attached == TargetPile.attached;
            };
            subtasks {
                1: TakeAndPutAll(SourcePile, TargetPile);
            };
        }
        // else
        {
            preconditions {};
            subtasks {
                1: TransportAllElsewhere(SourcePile, TargetPile);
            };
        }
    }

    method TakeAndPutAll(Pile SourcePile, Pile TargetPile) {
        // done
        empty {SourcePile.contains.size() == 0;};
        // else
        {
            preconditions {
                SourcePile.contains.size() > 0;
            };
            subtasks {
                G = SELECT(Agent, {G.type == "CRANE"; G.at == SourcePile.attached;});
                K = SELECT(Container, {K.top == SourcePile;});
                1: Take(G, K, SourcePile);
                2: Put(G, K, TargetPile)>1;
                3: TakeAndPutAll(SourcePile, TargetPile)>2;
            };
        }
    }

    method TransportAllElsewhere(Pile SourcePile, Pile TargetPile) {
        // done
        empty {SourcePile.contains.size() == 0;};
        // else
        {
            preconditions {
                SourcePile.contains.size() > 0;
            };
            subtasks {
                K = SELECT(Container, {K.top == SourcePile;});
                1: Transport(K, TargetPile);
                2: TransportAllElsewhere(SourcePile, TargetPile)>1;
            };
        }
    }

// ------------------------------------------------
// some multi-tasks goals

    // problem nav-c
    method NavC(Agent R1, Location L1, Agent R2, Location L2) {
        empty {R1 == R2;};
        {
            preconditions {};
            subtasks {
                1: Navigate(R1, L1);
                2: Navigate(R2, L2)>1;
            };
        }
    }

    // problem PbD
    method PbD(Container K1, Pile P1, Container K2, Pile P2) {
```

```
        empty {K1 == K2;};
        {
            preconditions {};
            subtasks {
                1: Transport(K1, P1);
                2: Transport(K2, P2)>1;
            };
        }
    }

    // problem all-c
    method AllC(Pile P1A, Pile P1B, Pile P2A, Pile P2B, Pile P3A, Pile P3B) {
        empty {P1A == P1B;};
        {
            preconditions {};
            subtasks {
                1: TransportAll(P1A, P1B);
                2: TransportAll(P2A, P2B)>1;
                3: TransportAll(P3A, P3B)>2;
            };
        }
    }

}

// ---------------------------------------------

timePart { priority = -4; }
```