

***Sofia University “St. Kliment Ohridski”***

Faculty of Mathematics and Informatics

Department “Information Technologies”

MSc Thesis:

***“Planning in Multiagent Systems”***

***Zlatina Lubomirova Marinova***

MSc Student in Informatics, N 41729

***Supervisor: Galia Angelova, Ph.D., Associate  
Professor***

Sofia, 24.07.2002

# Table of Contents

Preface .....	3
Chapter 1: Introduction .....	5
1.1 The concept of Agent .....	5
1.1.1 What is an Agent? .....	5
1.1.2 Related concepts .....	7
1.1.3 Examples of agents .....	8
1.2 Distributed Systems of Agents .....	9
1.2.1 Some definitions .....	9
1.2.2 Types of agents .....	12
1.3 The Common Planning Task .....	13
1.3.1 A sample domain .....	14
1.3.2 Actions and Events .....	14
1.3.3 Action Representation .....	15
1.3.4. The planning process .....	15
1.3.5 Action selection .....	16
1.3.6. Planning with Macro Actions .....	17
1.3.7 Goal Ordering .....	17
1.4 Specifics of the Planning Task in DAI .....	19
1.5 Specifics of Plan Representations .....	20
Chapter 2: Methods of planning in DAI .....	23
2.1 DPS and Planning for multiple agents .....	23
2.1.1 Basic assumptions .....	23
2.1.2 Examples of distributed problems .....	23
2.1.3 Task Sharing and Result Sharing .....	25
2.2 Approaches to Distributed Planning (DP) .....	30
2.2.1 Centralized Planning for Distributed Plans .....	30
2.2.2 Distributed Planning for Centralized Plans .....	31
2.2.3 Distributed Planning for Distributed Plans .....	32
2.3 Combining Planning and Execution .....	38
2.4.1 Detailed Description .....	41
2.5 Reactive Approach to Distributed Planning .....	45
Chapter 3 Planning Research Library .....	48
3.1 Purpose and Some Technical Aspects of the System .....	48
3.2 Modeling Agents .....	48
3.3 Modeling Systems of Agents .....	51
3.4 Implementing Multiagent Planning Methods .....	52
3.5 Modeling Environment .....	54
3.6 Auxiliary Concepts Modeled .....	54
3.7 User Interface Specification .....	56
Chapter 4 Conclusions and Future Work .....	57
Conclusions .....	57
Future Work .....	57
References .....	59
Appendix A: Planning Research Library Class Diagrams .....	61
Appendix B: Planning Research Library Source Code .....	63

# *Preface*

Distributed Artificial Intelligence (DAI) is a relatively new discipline of the Artificial Intelligence (AI) research field. There are two branches of DAI - distributed problem solving (DPS) and multiagent systems (MAS). MAS focus on modeling and simulation of societies and systems of agents. MAS are a rapidly evolving due to the efforts of researchers of different fields of AI and other sciences such as social sciences, NLP, cognitive science, etc. This wide interest in MAS is provoked by their vast application potential. They can be used for many scientific and practical purposes such as: modeling and study of biological systems, distributed problem solving, establishing societies of agents on the Internet, and many others.

Planning and coordination of agent plans are two of the most important issues that have to be considered by the designer of a MAS. Just like in classic planning, there is no single planning algorithm appropriate for all systems of agents. Many planning methods are described in multiagent literature but each of them is efficient for some and inefficient for other classes of agent systems or problems to be solved.

The purpose of this MSc thesis is to make a research of the existing planning methods and create an object library enabling easy modeling of multiagent systems and testing of planning algorithms. This library will allow comparing different types of multiagent planning for different task being executed by agents that are in different relations and environments.

Such a library will be very useful both from theoretical point of view – to enable modeling and visualization of the planning process in a MAS, and as a practical tool allowing to experiment with different planning methods before choosing the right one to be implemented in a future MAS.

This thesis is structured as follows:

- **chapter one** first introduces the basic concepts of agent, agent system and planning, and then briefly outlines the specifics of multiagent planning;
- **chapter two** describes some of the most popular methods for planning for multiple agents together with their implicit assumptions and limitations;
- **chapter three** is dedicated to the developed **Planning Research Library** . It describes the models of the basic concepts and planning

methods. A suggestion for development of software system using the **Planning Research Library** is presented at the end of the chapter.

- the **conclusion** chapter summarizes the results of the research and development of the planning library and discusses the possibilities for its future development;
- there are two **appendices** containing the class hierarchy diagrams (Appendix A) and the source code of the **Planning Research Library** (Appendix B).

# ***Chapter 1: Introduction***

Distributed Artificial Intelligence (DAI) is a subfield of AI which studies the interaction within systems of agents trying to solve a common problem or having to operate in the same environment. These agents can be of quite a different nature: computers, persons, animals, robots, etc. This chapter is intended to present a short overview of DAI and more specifically the problem of multiagent planning. We will first introduce the basic notions of agent and systems of agents. Then we will concentrate on the planning task, first introducing the general planning problem and then outlining the specifics of planning in DAI.

## ***1.1 The concept of Agent***

### **1.1.1 What is an Agent?**

There are many definitions of an agent in the AI literature. Here we will present some of them just to show that no consensus has been reached yet as to what an agent is. We will give some formal and a number of more informal definitions that can be found in the literature on agents.

As Hendler has stated in his IEEE Intelligent Systems editorial on agents (Hendler,1999, p.34), nowadays a lot of exciting research areas<sup>1</sup>, only some of which are related to each other, are using the term 'agent':

"pretty much anyone who is implementing anything, especially if it is distributed or network based, is willing to call it an agent. Thus, the term agent, even limited to an information technology context, refers to many different software concepts."

George Kiss (Kiss, 1996, p.247) provides an intuitive definition of agents as:

"Agents are the source of changes of states, i.e events. Action is the central characteristic of agents. ... The literature on agents appears to concentrate on clusters of issues such as having information, making decisions, and taking action".

Wooldridge and Jennings (Wooldridge,Jennings 1995, p.117) give a more formal definition. They present a weak and stronger notion of the concept 'agent'. According to the weak notion definition, the term agent is used to:

---

<sup>1</sup> Such as: decision making, mobile robotics, planning systems, interface animation, web search tools.

"denote a hardware or software-based computer system that enjoys the following properties:

- Autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;
- Social ability: agents communicate with other agents and humans via some kind of agent communication language;
- Reactivity: agents perceive their environment, and respond in a timely fashion to changes that occur in it;
- Proactivity: agents do not simply act in response to their environment, they are able to exhibit goal directed behavior by taking the initiative;"

To the above four characteristics/properties two more were added in the following years:

- "Temporal continuity: agents are continuously running processes;
- Goal orientedness: an agent is capable of handling complex, high level tasks. The decision how such a task is best split up in smaller subtasks, and in which order and in which way these sub-tasks should be best performed, should be made by the agent itself." (Hermans, 1999, p.3)

It is quite clear that these requirements are really weak and the simplest example of an agent meeting them could be the so called softbot ("software robot") - such as a software daemon (e.g. a background software process in Unix).

For researchers in the field of AI an agent is much more than what we describe above. They tend to expect agents to possess some quite human characteristics such as knowledge, beliefs, intentions, even emotions and benevolence. Agents in the AI domain are expected to demonstrate intelligent behavior and are usually called 'intelligent agents' as opposed to simple agents. Wooldridge and Jennings (Wooldridge, Jennings,1995, p.117-118) further develop the agent notion to meet these expectations by adding some other characteristics/requirements such as:

- "mobility: the ability of an agent to move around an electronic network;
- benevolence: the assumption that agents do not have conflicting goals, and that every agent will therefore always try to do what is asked of it;
- rationality: the assumption that an agent will act in order to achieve its goals and will not act in such a way as to prevent its goals being achieved - at least insofar as its beliefs permit;

- adaptivity: an agent should be able to adjust itself to the habits, working methods and preferences of its user".

Currently there is no agent implementation to meet the strong notion of agent, but there are many that possess some of these characteristics. No consensus has been reached yet as to what are the most important of the above characteristics for an agent. Still most researchers agree that these are the characteristics that distinguish ordinary programs from agents.

According to Ferber (Ferber, 1999, p. 9) a minimal common definition is:

"An agent is a physical or virtual entity

- (a) which is capable of acting in an environment,
- (b) which can communicate directly with other agents,
- (c) which is driven by a set of tendencies,
- (d) which possesses resources of its own,
- (e) which is capable of perceiving its environment (to a limited extent),
- (f) which has only a partial representation of this environment (and perhaps none at all),
- (g) which possesses skills and can offer services,
- (h) which may be able to reproduce itself,
- (i) whose behaviour tends towards satisfying its objectives, taking account of the resources and skills available to it and depending on its perceptions, its representations and the communications it receives."

### **1.1.2 Related concepts**

As we can see there are two key concepts common for most agent definitions. They are 'action' and 'environment'. We will now define them precisely in order to avoid misinterpretations.

Generally an agent has a set of actions it is able to perform and that set defines precisely how the agent can modify or at least influence its environment. Obviously not all actions are applicable in a particular situation and that is why each action has associated preconditions, describing the conditions for performing it. Usually actions are also characterized with uncertainty. That is, it is not guaranteed that same actions will have the same results when performed by the agent in identical situations. Furthermore, it is possible that an action that successfully changed the environment

once, will not have the same effect the next time or even fail to accomplish any desired change to the environment. Hence, we say that usually agents have partial control over the environment and rarely have complete control. The basic question facing the agent is which action to choose to perform out of the available set when trying to satisfy its goal(s). The complexity of this decision depends a lot on the properties and the characteristics of the environment occupied by the agent.

Now we will shortly list the main environmental properties as classified by Russell and Norvig. The basic characteristics of the environment are presented as pairs of opposite concepts (classes) as:( given in Wooldridge, 1999, p.30):

- Accessible/inaccessible – accessibility is defines whether an agent can get a comprehensive and accurate view of the environmental state and keep it updated. While it is easier to design agents for accessible environments, MAS deal mostly with inaccessible ones.
- Deterministic/non-deterministic – deterministic environments are those where every action has one predictable outcome at any given moment.
- Static/dynamic – we define an environment as static if it is changed only due to agents’ actions. Dynamic environments have some internal mechanisms of change that are beyond agents’ control.
- Discrete/continuous – discrete environments offer a limited number of actions and percepts to agents (e.g. chess game).

Environments that are inaccessible, non-deterministic, dynamic and continuous proved to be most difficult to design agents for.

### **1.1.3 Examples of agents**

As we already gave some examples of ‘simple’ agents, we will now focus on what we defined as ‘intelligent’ agents. Some examples are:

- Autonomous robots for sample gathering in space missions;
- Software agents that are programmed to search the Internet for information.

For the purpose of this thesis we will use the term agent when speaking of intelligent agents.

## ***1.2 Distributed Systems of Agents***

### **1.2.1 Some definitions**

The field of AI studying distributed systems of agents (DSA) is called Distributed artificial intelligence (DAI). DAI is concerned with understanding and modelling action and knowledge in collaborative enterprises. To make it more clear we should say that DAI is rather dealing with concurrent processes and not with parallel processing for improving efficiency. As Moulin says in his Overview of DAI (Moulin, Chaib-Draa, 1996, p.6), it does not address problems related to parallel computer architectures, parallel programming languages or distributed operating systems but distributed interpretation, distributed planning and control, cooperating expert systems, computer supported cooperative work, etc.

Ferber presents a more formal definition of distributed systems of agents or 'multi-agent system' as he calls it (Ferber, 1999, p11). According to it:

"the term 'multi-agent system' is applied to a system comprising the following elements:

- 1) An environment  $E$ , that is, a space which generally has a volume.
- 2) A set of objects,  $O$ . These objects are situated, that is to say, it is possible at a given moment to associate any object with a position in  $E$ . These objects are passive, that is, they can be perceived, created, destroyed and modified by the agents.
- 3) An assembly of agents,  $A$ , which are specific objects ( $A$  is subsumed in  $O$ ), representing the active entities of the system.
- 4) An assembly of relations,  $R$ , which link objects (and thus agents) to each other.
- 5) An assembly of operations,  $Op$ , making it possible for the agents of  $A$  to perceive, produce, consume, transform and manipulate objects from  $O$ .
- 6) Operators with the task of representing the application of these operations and the reaction of the world to this attempt at modification, which we shall call the laws of the universe."

An illustration of a system of agents is presented in Figure 1.

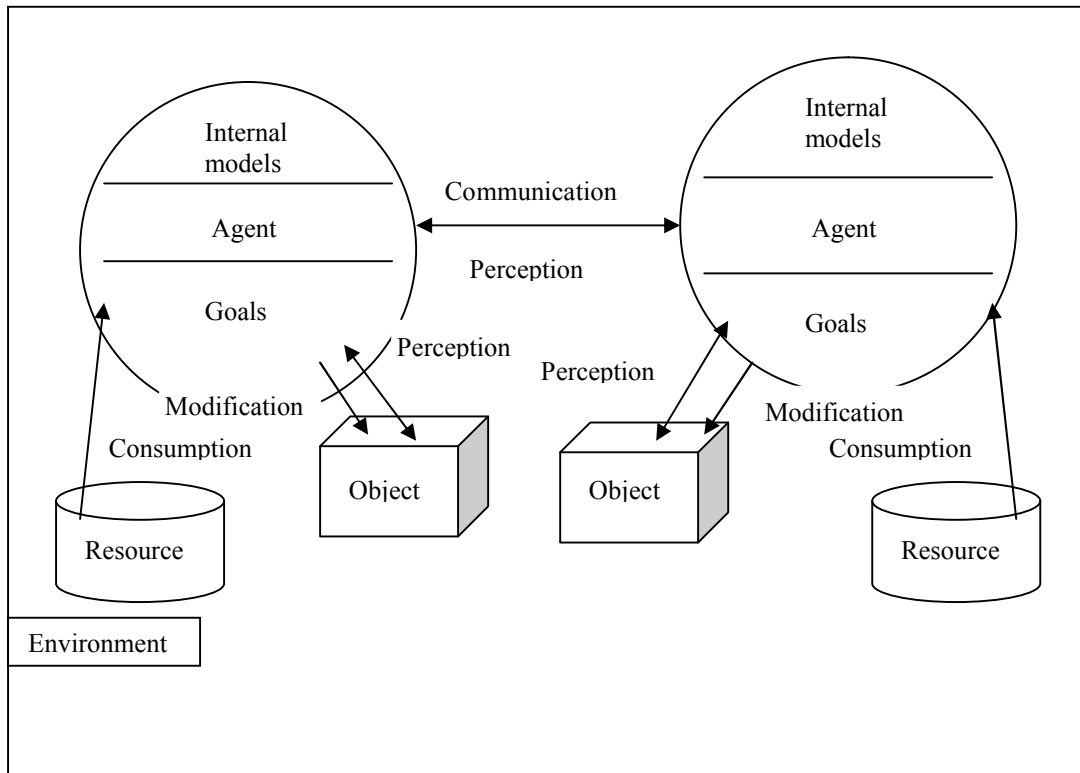


Figure 1, Illustration of a distributed system of agents

Some of the most important characteristics of distributed systems of agents are:

- Agents have limited skills and incomplete information;
- System control is distributed;
- Data is decentralized;
- Computation is asynchronous. (Weiss, 1999, p.3)

There are two branches of DAI - *distributed problem solving* and *multiagent systems*. *Distributed problem solving* (DPS) deals with dividing the task of solving a problem between a number of cooperating agents, that have a distributed knowledge about the problem and its evolving solutions. On the other hand research in the field of *multi-agent systems* (MAS) is mainly interested in the behavior of a set of autonomous agents solving a given problem.

According to Rosenschein (Rosenschein, 1993, p.797) the main difference between DPS and MAS is whether or not there is "a **single body** who is able, at design time, to directly influence the preferences of all agents in the system." DPS are centrally designed to solve a common problem so their designer may constitute the preferences of all agents in the system. MAS are also distributed systems but there is no single designer who stands behind all the agents. Each of the agents may have

different goals that may even conflict with those of other agents so there is no group notion of utility as in DPS. The agents in a MAS should be able to deal with other social phenomena such as cooperation and competition. The best a MAS designer can do is to design some aspects of the environment so that certain behavior of the agents is motivated. Rosenschein states that "this need for **indirect** incentives is one element that distinguishes MAS research from DPS research"(Rosenschein, 1993, p.798, Consenting Agents).

One may ask why we need to build a complex distributed system instead of using the standard AI problem solving mechanisms. There are several advantages of a distributed system over one single, centralized problem solver as pointed out by Moulin and Chaib-Draa (Moulin, Chaib-Draa, 1996, p.5):

- *"faster problem solving* - searching for solutions is parallelly performed;
- *decreased communication* - only high level partial solutions are exchanged between agents, instead of sending all raw data to a central site;
- *more flexibility* - agents can dynamically team up, thus combining different abilities while solving the problem;
- *increased reliability* - agents can substitute for each other if one fails."

On the other hand there are some disadvantages of the autonomous agents approach. The most significant ones according to Van Dyke Parunak (Van Dyke Parunak , 1996, p.144 in Foundations of DAI) are that:

- "theoretical optima cannot be guaranteed;
- predictions for autonomous systems can usually be made only at the aggregate level;
- in principle, systems of autonomous agents can become computationally unstable."

Currently there is no single methodology for analyzing, designing and implementing systems of agents. It is generally accepted, though, that each agent's internal knowledge and outgoing behavior should be modelled together with its interaction with other agents.

So far we saw that the main components of a distributed system are: the agents, the environment they populate and the interactions between them. By varying these three attributes, we can create different types of distributed systems of agents.

Next we present a table showing the range of each property of these components of a DSA.

	<b>Attribute</b>	<b>Range</b>
<b>Agents</b>	Number	From two upward
	Uniformity	Homogenous...heterogeneous
	Goals	Contradicting...contradictory
	Architecture	Reactive...cognitive
	Abilities	Simple...advanced
<b>Interaction</b>	Frequency	Low...high
	Persistence	Short-term...long-term
	Level	Signal-passing...knowledge-intensive
	Pattern(flow of data or control	Decentralized...hierarchical
	Variability	Fixed...changeable
	Purpose	Competitive...cooperative
<b>Environment</b>	Predictability	Foreseeable...unforeseeable
	Accessibility and knowability	Unlimited...limited
	Dynamics	Fixed...variable
	Diversity	Poor...rich
	Availability of resources	Restricted...ample

Table 1 Ranges of DSA attributed

## 1.2.2 Types of agents

Now let us pay special attention to the two basic types of agent architectures used in MAS that gave names to two classes of agents – *reactive agents* and *deliberative (cognitive) agents*. These types of agents are completely different from one another and so we will try to explain in details their properties, differences and what is their influence on the properties of the MAS they form.

*Cognitive agents* have an explicit symbolic representation of their environment and they use it to reason about possible events. *Reactive agents*, on the other hand, have no representation of their environment and are simply reacting to events in a “stimulus-response” pattern of behavior.

To make the differences between cognitive and reactive agents more clear let us introduce the notion of *cognitive cost*. It measures the complexity of the overall architecture needed when solving a problem. Cognitive agents naturally have a high cognitive cost since they are supporting a complex model of the environment and they have to keep it updated constantly. Reactive agents are on the other end of the scale - they have no internal representation of the world and are very simple as architecture. That is why their cognitive cost tends to what is called cognitive economy - the ability to perform complex tasks using a simple architecture.

The cognitive agents' complex architecture allows them to solve problems relatively independently. They are efficient when performing tasks even in small groups. Reactive agents on the contrary are designed to work in big communities and are helpless if isolated. Reactive agents are also *situated*, that is they do not consider past event and do not try to predict future ones. Agent's actions are based on what events happen in the world at the moment, on the way they percept and distinguish these events in order to react to them. Hence, reactive agents are incapable of planning ahead. This characteristic is as much a weak point as an advantage of reactive agents because they don't have to update an internal model of the world every time something unexpected happens. Thus, systems of reactive agents are more robust, flexible and stable than systems of cognitive agents, as even if one agent stops working, the rest will dynamically reallocate its tasks. This is due to the fact that roles are adopted dynamically in accordance to perceptions of current environmental needs.

### ***1.3 The Common Planning Task***

Now let us discuss the common task of planning as described in mainstream AI, its essence, existing methods and problems. First we will try to answer some basic questions such as : "What is a plan?" and "Why do we need planning?".

Plans in AI are frequently viewed as a sequence of actions that an agent, that is capable of changing its environment, should follow in order to achieve one or more goals. The process of generating a plan is called **planning**. The inputs essential for a planning process are: *initial world state*, a *set of possible actions* the agent can make to change the world, and a *set of goals* it wants to achieve. Planning is effective only in highly predictable environments, while in chaotic domains it is infeasible and the agent can only react to events.

Tradeoff should often be made between planning and reacting to events. While planning is reliable for controlling agent's behavior, it is not always fast enough to wait for in critical situations. On the other hand reaction is faster but it can lead to inappropriate or unsuccessful behavior. That is why many researchers adopt a hybrid approach when designing their agents.

Since there is much literature on planning we won't go into too much detail in this section but will only try to outline the most important topics. The following issues are of crucial importance to planning: *search control*, *action representation*, *goal*

*protection, time modelling, goals ordering.* To illustrate these concepts we will use the well known Blocks world model.

### 1.3.1 A sample domain

The Blocks world is a two dimensional world, consisting of a table of unbounded size and a number of square blocks, labelled with distinct letters, that can be arranged to form stacks. (an illustration of a Blocks world problem is presented in figure 2). Blocks world states are described by four kinds of literals:

(CLEAR x) - there is no block on top of x.

(HOLDING x) - a hand of an agent is holding block x.

(ON x y) - block x rests directly on block y.

(ONTABLE x) - block x rests directly on table.

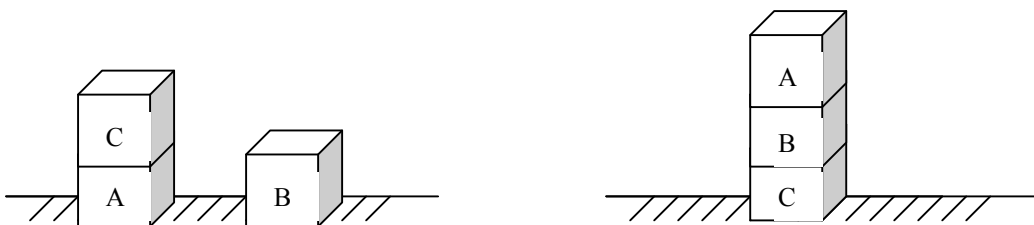


Figure 2a Blocks world problem initial state

2b Blocks world problem goal state

Any state description for a completely known world is defined by a conjunction of these literals. For example, the initial state in Figure 2a can be represented as:

(CLEAR C)(ON C A)(ONTABLE A)(CLEAR B)(ONTABLE B).

### 1.3.2 Actions and Events

Now lets say a few words about actions before we proceed to action representation. We assume that "an action is any change in the world state caused by an agent executing a plan" (Vere, 1992, p. 1160). Vere distinguishes two kinds of actions - *primitive* and *macro actions*. An action is classified as *primitive* if the details of its execution are of no importance to the planning task. A *macro action* on the other hand consists of primitive actions together with other macro actions, thus resembling procedures in programming languages.

Another important concept for planning is that of *event*. We should distinguish between actions and events. According to Vere (Vere, 1992, p. 1160) "events are triggered automatically because of natural or artificial mechanisms, whereas an action occurs only if an agent elects to perform it". Therefore, an agent is incapable of executing an event, it can only take actions to achieve a state of the world that will trigger that event.

### 1.3.3 Action Representation

A generally accepted method of action representation is the one using *preconditions* and *postconditions*, both of which are conjunctions of literals. *Preconditions* define the conditions that should be true in the current state of the world in order to perform an action. *Postconditions* represent the conditions that will be valid after the action is performed. If we need to model changes in the world that will occur when the action is performed, then this is not a primitive but a macro action that should be further decomposed. So an action is defined as:

```
(<action name> <action-variable-list>
  <preconditions>
  →
  <postconditions>)
```

The same model can be used to represent simple events. Their preconditions will contain the causes of the event, and postconditions will show its effects. Events can be inserted into plans using the same backward-chaining mechanism as that described later on in this chapter. There is only one difference between actions and events in this representation and it is these event rules that must be allowed to chain forward whenever their preconditions are satisfied.

### 1.3.4. The Planning Process

Classic planners keep their goals on a stack and do an ordered depth-first search through the space of possible plans. A plan consists only of primitive actions. Goal protection is implemented using assertions that signify which literals should not be contradicted. Goal protections according to Vere (Vere, 1992, p. 1162) are used to bind together the elements of a plan and to ensure its coherence. If there are no goal protections the planner may achieve one goal at the expense of another.

*Node expansion* (or backward chaining) is the process of inserting an action in the plan that will achieve a goal that is not yet true. It selects an action which postcondition matches the goal and inserts its preconditions as new goals (subgoals). Fortunately not every goal requires node expansion (otherwise the planning process would never terminate). *Linking* is another way of achieving a goal. It is performed when the goal is matched to another assertion in the plan. In fact linking is always tried first and only if it fails node expansion takes place.

However not every precondition of an action should be expanded. For example, if we have the precondition (ON upper-block lower-block) in the representation of an unstack action, it should be achieved only by linking because it is pointless to stack two blocks in order to unstack them. Such preconditions are called "hold" conditions. Nowadays planners have syntactic means to specify which preconditions are not appropriate for node expansion.

If at some point of the plan generation a conflict is detected, another operation, called *conflict resolution*, should be applied. Vere (Vere, 1992, p.1166) defines a conflict as "the situation where two unordered nodes assert contradictory literals". For example, one node of a Blocks world planner may assert (CLEAR C) and another one (not connected with the first) may assert (NOT (CLEAR C)). This conflict can be resolved by ordering these two nodes with respect to the protection relations of their upper nodes. Conflicts should be resolved as soon as they appear because otherwise it is not certain whether an assertion is true at a particular step of the plan. That makes linking impossible and hinders the process of selecting a best node for expansion (described below).

We have just described the three major operations of a basic planner: node expansion, linking and conflict resolution. The process of plan synthesis is merely a repetition of these operations. A plan is considered to be complete when all conflicts have been resolved and all goals have been removed from the stack. If at some point it is not possible neither to link a particular goal nor to expand a node, the planner has to backtrack up the search tree and substitute one of its previous choices.

### **1.3.5 Action selection**

As we have seen above when node expansion is performed there can be more than one appropriate action. In that case the planner has to decide which one to try

first. To select one it can use either *domain-independent criteria* or *domain specific advice*. Some commonly accepted *domain independent criteria* for ordering candidate actions are:

- the number of nonlinkable preconditions - the more preconditions that can be linked at a single step the less work remains to be done (less node expansions will have to be performed);
- the number of "bonus goals" - if by chance an action achieves not only the top goal in the stack, but also some additional one, these goals are called bonus goals (satisfied simply by linking). As the agent obviously tries to generate a simpler plan if two actions have the same number of nonlinkable preconditions, it will be better to satisfy the one with more bonus goals.
- resource consumption - actions may also differ in the amount of valueable resources (e.g. time, money) they will need.

*Domain dependent criteria* can be implemented by adding additional preconditions. For example, in the Blocks world domain it might be profitable to rate the possible actions of lifting a block on the basis of weight. However this approach is not appropriate for complex domains. In that case a human operator may be asked to make the choice, thus reducing the possibility of the planner getting lost in search.

### **1.3.6. Planning with Macro Actions**

Often it is convenient to group sequence of actions into macro actions and moreover this can reduce search. Again goal protections are used to synchronise macro actions. In planning with macro actions both goals and action nodes can be expanded either into macro actions or primitive actions. Practically the designer is the one who plans, not the planner. Also plans constructed by macro actions are less flexible, as the planner is simply repeating and combining only predefined sequences.

### **1.3.7 Goal Ordering**

The planning process depends on the order in which goals are selected for achieving. This is due to the changes that previous actions might have made in the world state. Some improper ordering may cause searching to last longer or even to end up with no solution found. There are several solutions to these problems.

One obvious solution is to try to reorder the goals whenever a solution is not reached. Unfortunately this leads to big computational expenses so instead an hierarchy can be established on the literals of a domain. The most difficult literals for achieving are situated at the top while easily reproducible ones are at the bottom of the hierarchy. The goals in the planner stack are reordered so that higher level ones are achieved prior to satisfying lower level goals.

Goal protection may also cause problems and even prevent us from finding a solution for all possible goal orderings. The only solution in such situations will be to violate goals that are already achieved and then re achieve them. Vere (Vere, 1992, p. 1168) has proposed a procedure called *plan splicing* that can be used for conflict resolution and as an alternative to permutation of goals upon failure as well. Splicing includes violation of protected goals and recursive deletion of parts of the generated plan. Vere has shown that splicing is efficient even at execution time as a reaction to unexpected events.

## ***1.4 Specifics of the Planning Task in DAI***

Now that we have shortly discussed the basic principles of a general planner, let us see what are the specifics of planning for systems of agents.

While planning for a single agent is the process of generating a sequence of actions for an agent based only on its goals, capabilities and the environmental constraints, planning in a system of agents should consider all constraints placed by all agents. The selection of an action is influenced by the agent's commitments to others, the activities of other agents that may change the environment and the hardly predictable evolution of the world caused by unmodelled agents (Durfee, 1996, p. 232).

Bond and Gasser (see Moulin, Chaib-Draa, 1996, p.27) indicate that "we can achieve greater coordination by aligning behavior of agents toward common goals, with explicit division of labor. Techniques such as centralized planning for multiple agents, plan reconciliation, distributed planning, organizational analysis are ways of helping to align the activities of agents by assigning tasks after reasoning through the consequences of doing these task in particular orders."

A plan that has been generated for multiple executing agents is called a *multiagent plan* and the process of its construction is called *multiagent planning*. The plan itself may be built up by one or more agents.

If one agent creates the multiagent plan, we are speaking of *centralized multiagent planning*. There are different ways to implement centralized planning. Cammarata has adopted an approach in which all agents first negotiate to choose a single coordinator that constructs the plan and all the rest have to follow it. Georgeff on the other hand proposed a solution in which all agents plan for themselves and there is a predefined central agent that collects their individual plans and tries to reconcile any conflicts (see Moulin, Chaib-Draa, 1996, p.27)

If more than one agent creates the multiagent plan, the process is called *distributed multiagent planning*. In that case there is a group of agents or each agent alone that generate the plan. This approach is usually used when there is no agent with a global view of the group activities. Von Martial (see Moulin, Chaib-Draa, 1996, p.27) has defined two classes of subproblems in distributed planning:

- *task driven planning* - when there is a common initial goal or task that has to be decomposed into subgoals and then distributed between planners, and
- *plan coordination* - when there are preexisting plans that have to be coordinated in order to avoid conflicts.

Distributed planning needs means for synchronization of plans. This synchronization may take place at several points: during task decomposition, at the time of plan generation or after the plan is created. Conflicts may appear due to incompatible states, incompatible order of actions or incompatible use of resources. There are several methodologies developed to resolve conflicts that will be discussed later on.

## ***1.5 Specifics of Plan Representations***

Synchronization of plans in DAI requires agents to be able to exchange goals, tasks, solutions and plans. Formalisms and protocols that are specific for multiagent planning have been developed over the years. Nowadays agents communicate mostly using one of the following alternatives:

- Contract Net
- Speech acts
- KQML

All of the above means are aiming at enabling particular conversations between agents such as: asking an agent for a service or information, responding to an agent's request and so on. They are designed for use in all kind of domains, by different kinds of agents and therefore they tend to be generic. Thus, protocols generally specify the kinds of behavior a message is supposed to provoke in an agent and do not engage in any definitions of the content of that message. Still, to be able to work together agents need to be able to understand each other completely, so the designer of a multiagent system has to define certain content definition language. Defining a content language is difficult as it has to serve many purposes and answer to many criteria. Durfee has summarized the requirements to a planning content language as follows:

“A planning content language needs to satisfy all of the constituencies that would use the plan. If we think of a plan as being comprised of a variety of fields

(different kinds of related information), then different combinations of agents will need to access and modify different combinations of fields. In exchanging a plan, the agents need to be able to find the information they need so as to take the actions that they are expected to take in interpreting, modifying, or executing the plan. They also need to know how to change the plan in ways that will be interpreted correctly by other agents and lead to desirable effects.” (Durfee, 1999, p. 150)

Currently there is no general plan representation language and designers define such languages for each particular domain. It is generally accepted that in a particular multiagent system agents are using identical plan representations and are designed to interpret them in the same way. Nevertheless, there are some attempts at defining a common plan representation:

- STRIPS operators in their original form, or augmented in some ways, are used quite often because of their simplicity and yet are appropriate only in certain domains;
- SRI has developed a plan representation language for their Cypress system. This language is using a more general description of plan based on definition of ACTs as is described below;
- Petri Nets are also used as operational formalism for plan representations. There agents need not have the same internal plan representations as long as they interpret plans in a way that is consistent with the operational semantics.

Let us now describe in details the Cypress plan representation (see Durfee, 1999, p.150-151). ACTs consist of the following fields:

1. Name –a unique label;
2. Cue – the goals that ACT can achieve;
3. Precondition – necessary conditions that should hold in the world before ACT is performed;
4. Setting – features of the world-state that are bound to ACT variables;
5. Resources – what resources are used while ACT is executed;
6. Properties – other properties associated with ACT;
7. Comment – documentation information

8. Plot – partially-ordered sequence of goals or actions to be executed.

As this is a general plan representation language it does not provide strict definitions of contents of each of the above fields.

## ***Chapter 2: Methods of planning in DAI***

### ***2.1 DPS and Planning for multiple agents***

Distributed planning has a lot in common with distributed problem solving. On the one hand while a distributed problem is solved, each agent has to create a plan of its own activities and then to coordinate their actions to avoid negative interactions, thus creating a multiagent plan. On the other hand, planning for multiple agents can be regarded as a distributed problem they have to solve. That is why we will next present an overview of the specifics and techniques of distributed problem solving.

#### **2.1.1 Basic assumptions**

When a system for distributed problem solving is build some presumptions are made about the main properties of its elements. Some of them are as follows:

- tasks, resources and/or skills are irregularly distributed among agents, causing agents to work together in order be more effective or even to accomplish their tasks at all;
- agents are designed to be able to work with others and are benevolent and honest;
- there are some problems that need to be solved and some requirements their solutions have to meet;

#### **2.1.2 Examples of distributed problems**

Distributed approach can be applied for solving many types of problems (including planning). It usually improves the solution or execution time due to parallelism techniques exploited. There are many problems that are distributed in their nature – agents need to share resources, knowledge and data while solving the problem, or they will benefit from combining their efforts in solving that problem because of inherent distribution of capabilities or beliefs (and knowledge) among them. Another motivation for using DPS is that the result (usually a plan) should be distributed among the agents to act upon it. Next we will present some examples of distributed problems that will be used in the rest of this thesis as sample domains for planning methods and algorithms.

### *Tower of Hanoi(ToH)*

*Tower of Hanoi* is a well-known problem in the AI discipline and is widely used as illustration of problem solving techniques. Its world consists of three pegs and a number of discs of graduated sizes. Possible actions include moving a disk (one at a time) from peg to peg without placing larger disks on smaller ones. The goal state usually is to move all the disks from the initial peg to a given one.

ToH is an example of the class of problems that allow for parallel planning techniques to be used to decrease the time for finding solution.

### *Distributed Sensor Network Establishment (DSNE) and Distributed Vehicle Monitoring (DVM)*

These are typical examples of problems where capabilities, expertise and data are distributed among agents. The task is to monitor a large geographical area (DSNE) and track vehicle movements (DVM). Obviously, it is impossible for a single agent to monitor the entire region, so the task should be decomposed and distributed among a network of agents, each of them responsible for part of the monitored area. After a vehicle is monitored, an attempt has to be made to track its trajectory. It will be very costly (in terms of time and resources) to solve this problem using a centralized agent that receives data from the DSNE. Both time and communication expenses can be decreased by designing each of the monitoring agents to be able to interpret sensed data and then communicate their hypotheses of vehicle track.

### *Distributed Delivery (DD)*

The DD problem is defined as having to deliver objects between locations using several agents that can act in parallel. In order to avoid conflicts and to enable coordination, a multiagent plan has to be generated, followed and in cases of unforeseen events - updated.

### *Internet Downloading Domain*

In this domain each agent is given a set of documents it has to download from the Internet. Each agent is also eager to minimize the cost of downloading and therefore agents can benefit by communicating the documents they need, and then downloading each document only once and sharing it.

### 2.1.3 Task Sharing and Result Sharing

There are two mechanisms for collaboration between agents when a distributed problem is solved. In order to benefit from the relative autonomy and independence of agents, collaboration is based on sharing tasks and results.

#### *Task Sharing*

*Task sharing* naturally uses the inherent decomposition of the solved problem and the ability of agents to work in parallel. When an agent realizes it is overburdened with tasks, it first tries to decompose its task(s) to subtasks that can be handled by other agents. Then the agent has to allocate the resulting set of tasks to agents it knows that possess needed skills. Next each of the agents accomplishes its subtask(s), using recursive task sharing if needed, and when finished it passes the results back to an agent that composes the final solution to the problem (usually the agent that first decomposed the problem, since it is very likely that it will know how to synthesize results).

Sharing tasks is easy when agents are with identical capabilities – such agent systems are called *homogenous*. Then an overburdened agent randomly allocates some of its tasks to the idle agents in the system. Unfortunately, in most systems agent capabilities are far from identical – such systems are heterogeneous. Problems too, rarely can be decomposed into similar subtasks. As in human societies it proves to be useful to have different agents specialize in performing different tasks. Thus, agents are easier to build and train and there is no loss in terms of capabilities – when at a given moment only a small part of the capacities of a ‘multipurpose’ agent are utilized.

Sharing tasks in heterogeneous systems when solving a problem that cannot be decomposed to equal sub-problems requires using of more intricate mechanisms. In the simplest case, an agent should keep information about the skills each of its acquaintances possess and then simply select one of the agents capable (as far as it knows) of completing a given task from the set and assign it to him. Unfortunately, it may appear that some of the agents are already busy performing other tasks and it is better to realize that prior to assigning them another one. This can be accomplished by using the Contract Net protocol with directed contracts or focused addressing based on the capabilities table. In that case the manager (the agent to announce a task) requires bidders (agents that respond to the announcement) to declare

acceptance/availability (Durfee, 1999, p.127) in their answers. If it turns up that none of the acquaintances is free at that moment, there are several options for the announcing agent (manager) as described by Durfee in (Durfee, 1999, p.128-9):

- Broadcast contracting - the task will be broadcasted to all agents present in the environment, not only to the ones the agent is aware of as in focused or direct contracting. It can be used when the manager wants to update its capabilities table – it can never be sure the table contains all agents present in the environment.
- Retry – the manager can keep announcing the task periodically until someone becomes free. An important issue then becomes how often to re-announce tasks as not to overload the system and yet utilize agents free time effectively. A possible solution is to make the protocol work completely differently so that agents announce their availability and managers bid with pending tasks.
- Announcement revision – the manager may decide to change the eligibility specification (which is part of the original message) so that more possible contractors are reached. It could even be an iterative process until a bid for contract appears.
- Alternative decompositions – sometimes it may prove feasible for the manager to try decomposing the task in another way so that other contractor agents can be addressed with the new set of subtasks.

### *Result Sharing*

Results are intermediate and final products of solving a problem by a particular agent. Therefore agents with different capabilities, knowledge and representations of the world may generate different results while performing the same tasks. Thinking of agents as intelligent creatures solving given problems (like scientists themselves) brought about the idea of making agents cooperate by sharing the results of their work. Agent performance at group level can benefit in the following ways:

- Confidence is increasing – having several agents arrive at the same results when working on the same problem, increases the certainty of the correctness of this result.

- Completeness – having each agent share with others the results of all tasks it has completed, provides the group of agents (and hence each of them) with a more complete solution to the global task.
- Precision – having more complete view over the overall task allows agents to refine the solutions to their own subproblems. This idea is widely used by the PGP algorithm for problem solving as we will see later in this chapter.
- Timeliness – exchanging results allows agents to work effectively in parallel and minimizes the time spent in redundant solving of subproblems by several agents.

In order to enjoy all these advantages of sharing results agents have to be both competent and “prudent”. They need to be competent in interpreting received results and they also should be very selective and find the right balance between sending all results (that is costly and can bog the system) and exchanging too little information to help others in their work. Agents achieve the integration of exchanged information by treating all the results as partial and tentative. Thus they often have to create interpretations of combining or contradicting hypotheses. To ease that process usually agents are designed and grouped to have similar representations of results. Then it is mostly a matter of exchanging enough and proper results (and time) to eventually build an overall solution. That brings forward the question of selective communication – what and how often to exchange with other agents. Another consideration is as how to regard the received information in respect to its credibility and priority compared to own results.

There are several strategies to addressing these issues:

- Use a **shared repository** (such as a blackboard) instead of propagating tentative results to all agents. This approach is very useful when agents have no idea who may be interested in their results.
- **Distributed Constraint Heuristic Search (DCHS)** is another option for agents to deal with limited resources. The idea is to create a resource agent that schedules and manages the use of its resource. Then there are two strategies for consumer agents (those who use resources) to contend for resources – use market mechanisms such as auctions or to address resource agents with their tentative demands and

then let them solve the constraint-satisfaction problems of scheduling their resources. The second alternative proves to be very efficient because communication is limited and focused, and because aggregating tentative demands allows resource agents to avoid useless backtracking while scheduling resources.

- Create an **organizational structuring** as a means for reducing communication. In the DVM task for example, it is reasonable for agents to communicate much more with their direct neighbors than with agents that monitor distant regions. The main idea behind this strategy is ‘that agents have general roles to play in the collective effort, and by using knowledge of these roles the agents can make better interaction decisions’ (Durfee, 1999, p.135). The organizational structure is generally defined by the agents’ roles, preferences and responsibilities. Roles describe the types of tasks an agent will be expected to perform and with some priority rules to help it work efficiently in multitask situation. Responsibilities are that really help reduce communication as each agent is informed only of results that aid it perform its tasks better - that is, responsibilities actually show to agents whom to inform of what. They also impose the belief relationships between agents, i.e. what to believe and to what extent when receiving information from others. There are several ways of creating an organizational structure – it can be based on the problem decomposition structure while tasks are allocated; it can emerge in a bottom-up manner as result of negotiations between agents; or be imposed as a top-down design solution. Obviously in an open multi-agent system the organizational structure can be (or become) too big/wide and actually slow down the overall performance when each agent is trying to keep it updated. Fortunately, it is practically enough for each agent to know only a local portion of the organization, that specifies its own location and its connections with other agents.
- Different **communication strategies** can be used in order to minimize the redundant information send. First, even if some recipient may be interested in a result generated by an agent, it can be useless to it at a

given time interval (when it is still far from the point of using this result) or cause a distraction (the recipient starts exploring the same area of the solution space). Therefore, it makes sense to wait until an agent has reached local solution (it cannot proceed with it any further), but then finding the overall solution can be delayed and a lot of worthless hypotheses can be explored (before negative feedback from other agents is received). Depending on the design of the agents and the communication channel, and on the nature of the problem to solve, all kind of communication strategies can be used – from announcing all tentative results to sending only locally complete solutions, and anything in-between those two extremes. For example, in the DVM problem it will be more useful for the agents to exchange some preliminary hypotheses early on, in order to minimize the time spent in exploring tracks that are impossible according to neighbor's interpretation of sensed data. Yet another alternative for agent communication is to wait for results to be requested by others before sending them. In that case an agent can send a request for results when it can go no further in solving its problem or send pending goals it cannot realize. Finally, a communication strategy should determine how the problem of lost messages (containing results or requests) should be overcome. It can be decided that acknowledgement should be send after receiving a message (that can really bog the channel), and if it is not received the sender will repeat the original message. Another option is to try to predict how the receiver would react to the message and check whether the agent exhibits such behavior.

- Agents can easily deal with communication problems if **task relationships** are introduced. Task relationships denote the non-local effects that actions of one agent may have on those of others. Generally four types of relations are defined – *enable*, when the result of one agent actions is needed for some other agent to perform a task of its own; *facilitate* – the results of an agent help other agent(s) provide better solutions, or get them faster; *inhibit* – as opposed to enable; and *hinder* – as opposite to facilitate. Coordination can be significantly improved by using such relationships. For example, if an

agent discovers that one of his actions is enabling for a task of another agent, it may decide to do it early on so that the other agent can get to work earlier. On the other hand if an action is inhibiting or hindering another action then the agent may decide to wait until the other agent has finished up with his task so that not to delay its work.

All the strategies we have discussed so far are working well in systems with relatively static organizations with simple non-local task relationships. To be able to deal in complex domains with complex interrelations between actions, agents need to be capable of analyzing the current situation and generate a plan for communicating and interacting with other agents. We will next focus on these issues.

## ***2.2 Approaches to Distributed Planning (DP)***

Generally, planning can be regarded as problem solving task of creating a plan. Just like in distributed problem solving, both the planning task and the results from planning can be sources of distribution in a multiagent system. When the planning process is distributed among a multitude of agents, we are talking of **distributed plan formation**. If the result of the planning process (which is a plan) is further distributed for execution by several agents, it is called a **distributed plan**. The term distributed planning is used to describe situations when agents participate in distribute plan formation or act after a distributed plan, or both. In the following section we will discuss techniques for distributed planning that differ from the ones used for distributed problem solving.

### **2.2.1 Centralized Planning for Distributed Plans**

Centralized planning is used when there is a single agent that should for some reason (it is the only one capable, or having enough information) create a plan. That plan has to be a partial order one in order to be distributable among agents. Either the agent that generated the plan or a predefined (or a negotiated) coordinator agent is then breaking the plan into pieces (threads) that can be possibly executed in parallel and adding to each thread some coordination actions when needed. Threads are distributed between agents using some task allocation technique and agents act on the subplans they receive. Assuming that agents follow the plan strictly and have correct knowledge and predictions of the world and their actions, a state of the world in

conformance with the initial goals is achieved as a result of parallel execution of subplans.

Obviously to have the centralized planning technique work efficiently, it is most important that the most effective decomposition and distribution of tasks is found. It gets very tricky because availability of agents capable of achieving a subplan cannot be guaranteed at any moment. Therefore several iterations of decomposing and distributing a task may be necessary before agents actually start to follow a distributed plan.

Communication infrastructure appears to be decisive of the size of the subplans in the decomposition set. As an example, in the case of slow or unreliable communication it will be more efficient to have relatively independent subplans, that require minimum coordination and can followed by a smaller number of agents. In other words: 'there is some minimal subplan size smaller than which it does not make sense to decompose a plan. In loosely-coupled networks, this leads to systems with fewer agents each accomplishing larger tasks, while in tightly-connected (or even shared-memory) systems the degree of decomposition and parallelism can be increased' (Durfee, 1999, p.140).

### **2.2.2 Distributed Planning for Centralized Plans**

Distributed planning can take place even when the resulting plan is to be executed by a single agent. Distribution may be necessary due to lack of capability or knowledge (specialization) of the acting agent or for efficiency reasons. Therefore, distributed building of centralized plans can be regarded as a specialization of distributed problem solving techniques. Both task-sharing and result-sharing can be used here as a means for cooperation. At first, the problem (a goal) has to be decomposed and distributed among the participating planning specialists using some task-sharing method. Then coordination can be achieved either by exchanging a single partially-specified plan or by result sharing. In the first case each planner tries to modify and expand (extend) the partial plan it receives in accordance with its goals and it also ensures the plan remains consistent. If a planner cannot augment the plan for some reason (probably a dead-end for its actions) or it realizes that it is inconsistent, a backtracking procedure is started to enable finding of alternative plans.

When result sharing is used, planners try to generate partial plans in parallel and then merge them to form the controversial plan.

Distributed planning for centralized plans is successfully used in complex domains such as manufacturing, logistics and mission planning for unmanned vehicles.

### **2.2.3 Distributed Planning for Distributed Plans**

The most complex configuration for distributed planning is the one when both the process of planning and the execution are distributed. As this is the general case of planning in a multiagent system of cooperative agents, it may appear that none of the agents is aware of all the other agents and their behavior. Thus, it may be impossible and unnecessary to try and generate a complete multiagent plan, containing all the actions of all the agents in the system. On the other hand, it is preferable to have as minimum conflicts between agent plans as possible and even to enable cooperation by having agents help each other in their tasks (if it is possible and rational). Therefore, agents should be aware of the distributed plans of others and they should be capable of resolving conflicts and finding out of inter-task relationships. Some of the most popular techniques dealing with these problems are:

- Plan merging;
- Iterative plan formation;
- Negotiation.

#### *Plan Merging*

Plan merging is not interested in the distribution of goals – they can be allocated to agents in using a task passing mechanism or agents can be assigned tasks separately from each other (e.g. in the DD domain each robot receives its tasks directly from users). This technique is used when individual agents create plans for their activities without discussing them with other agents. Then/afterwards agents communicate in order to coordinate their action sequences in a way that no conflicts appear.

Here we will outline the algorithm of plan merging by a single agent – *centralized plan merging*. Let us assume that each of the agents has generated its plan on its own (and this plan is complete and consistent), using some planning method. Next all the agents deliver their plans to a single agent (a coordinator agent) that will

try to merge them and will add constraints on them (if needed) to ensure that conflicting actions are not performed at the same moment. So the basic problem the coordinator has to solve is to find out what are those sequences of actions in the plans of different agents that could cause a conflict. Generally, this problem requires using reachability analysis, i.e. having the initial state and a set of planned action sequences that can be performed asynchronously, find the states of the world that can be reached, enumerate the ones that include conflicts and avoid them by placing constraints on the action sequences. As the task of inferring all possible states of the world can be intractable, some heuristics should be used to narrow the search space. We will next present an algorithm suggested by Georgeff (as described in Durfee 1999, p.142-3).

This method requires several assumptions to be made:

- agents know all possible initial states of the world;
- each agent has built a correct, totally-ordered plan comprising of actions  $a_1$  to  $a_n$  so that  $a_1$  is applicable to any of the initial states,  $a_i$  can be performed in any of the states resulting from  $a_{i-1}$  and  $a_n$  accomplishes the agent's goal;
- actions are represented as STRIPS operators augmented with 'during conditions' that describe changes to the world that are valid only during the actions.

The assumption of using STRIPS operators significantly narrows the search space of interactions between plans. Since the effects of an action sequence are just the product of the effects of each of the actions, it is enough to identify of all possible states (that are results of combinations of the order of performing different action sequences) only those where particular actions interact (possibly causing conflicts).

The merging algorithm is analyzing the plans of a multitude of agents by examining pairs of actions that two agents can take at the same time. Let us say, for example, actions  $a_i$  and  $b_j$  are to be analyzed next as actions that agents A and B can take at the same time (plans of different agents are asynchronously executed). There are several possible configurations of dependencies between  $a_i$  and  $b_j$ :

- *independency* – two actions can be executed at the same moment of time if their preconditions, during conditions and postconditions are

satisfiable at the same state of the world. Such actions are called **commutable**;

- *precedence* – if the state of the world before any of those two actions has taken place modified by the effects of one of the actions (say  $a_i$ ) satisfies the precondition of the other action ( $b_j$ );
- *conflict* – if the neither action can precede the other.

Having derived dependencies between actions, we can proceed by analyzing situations safety. We define *unsafe situations* as follows:

- the situation of executing  $a_i$  and  $b_j$  together is unsafe if they do not commute;
- the situation of starting  $a_i$  before  $b_j$  is unsafe if  $a_i$  does not precede  $b_j$ ;
- the situation of beginning  $a_i$  and  $b_j$  is unsafe if any of its successor situations is unsafe;
- the situation of beginning  $a_i$  and ending  $b_j$  is unsafe if the situation of ending  $a_i$  and ending  $b_j$  is unsafe;
- the situation of ending  $a_i$  and ending  $b_j$  is unsafe if both of its successor states are unsafe.

The situation safety analysis is significantly narrowing the interactions search space thanks to independent actions. Obviously if an action commutes with all the others, it can be excluded from the interactions search space as not causing a conflict. This is particularly helpful in loosely-coupled MASs where each agent is working on its own and the majority of actions are independent. In that case plans to be merged are reduced to short sequences that can be analyzed. The algorithm continues by defining the space of unsafe situations (which can be an exponential problem) and then placing synchronization restrictions in agent plans. Restrictions usually suspend an agent's activity while there are ongoing actions of other agents that conflict with the next action of the agent.

The algorithm of plan merging is more complex when dealing with plans for temporal goals. For example, in the DD domain the agent's goal is not simply to deliver an object but to do it in a specified time interval (or at least by a certain deadline). In such applications plan merging is based on scheduling activities during fixed time intervals. Agents build their plans as independent schedules of their actions that can be in precedence ordering (some action has to be finished before some other

can be undertaken). Then the process of plan merging becomes a **distributed constraint satisfaction problem** (DCSP) of finding an agreeable schedule for agent tasks under the constraints of their precedence order, deadlines and resource consumption.

Plan merging can be performed in decentralized way as agents follow the main steps outlined above – goals are identified, local plans are generated and exchanged, then plans are merged using time and messages as means of coordination and conflict resolution.

#### *Iterative Plan Formation*

As we have realized above *Plan merging* can be used to enable parallel plan generation and execution with the assumption that each agent generates its plan considering only its goals. But in practice local decisions and plans are seldom independent of other agent actions. A bunch of approaches is developed to deal with situations where agents should consider global constraints and should be aware of possible actions of others while generating their plans. All of these approaches use coordination techniques during the process of plan formation instead of afterwards (as in plan merging). As a result agent plans are formed in iterations of subplan generation and coordination activities and hence we are speaking of methods of iterative plan formation.

One method is the **plan combination search** proposed by Ephrati and Rosenschein. First each agent is generating a set of all feasible plans that realize its goal(s) and then agents are engaged in searching through the space of all plans to find those that can be combined together to avoid conflicts. The states of the world and the results of agent actions are represented as sets of propositions. Given a current proposition set as a world state, each agent presents the changes to that set that will result of a single action of each of their plans. All changes are combined to produce possible next states of the world, which are then ordered using A\* heuristics. The best state is chosen and the process repeated until no agent has further propositions. In this method coordination is achieved during the process of refinement of plans as an agent is naturally constrained of acting when the chosen next world state does not include results of its actions (probably because they were in conflict with some other agent goals).

Corkill suggested using hierarchical planning for multiple agents – the so called **distributed hierarchical planning approach**. He illustrated the approach in

the Blocks world and added a “decompose plan” critic whose purpose is to discover conjunctive goals to be distributed. When conjunctive goals are allocated to agents, each agent receives a copy of the plan network as well. From that point on each agent keeps a model of other agent(s) plan(s) and updates it whenever it has information on the plans modifications. In the process of detailing plans to more comprehensive levels agents decide when to communicate to others about the changes they intend to make to the world state. Thus, each agent can detect conflicts of its actions with the effects that other agents plan to have on the world. The agent that discovers a conflict may ask the other agent(s) to WAIT until it is safe to perform the action causing the conflict. The algorithm terminates when a synchronized set of detailed plans is reached. Just like hierarchical planning in mainstream AI, distributed hierarchical planning proves to be very efficient because coordination and interactions can be resolved early on in the planning process (with more abstract forms of the plan).

Another version of a hierarchical planning approach allows agents to store planned behaviors at multiple abstraction levels and to decide on the level at which conflicts should be resolved. Thus this approach is based on a **hierarchical behavior space search** in/through the space of local plans of certain abstraction level. At/on each iteration of the protocol agents specify whether conflicts should be resolved at the current level or the planning process should continue to detail activities. Then a distributed constraint satisfaction search is performed to work out all conflicts. The termination of the algorithm is ensured by the facts that each plan can be abstracted to finite number of levels (at some point a complete plan is reached) and agents can make a finite number of changes to their plans without getting to a planning cycle. The tricky tradeoff that should be made in this approach is whether to resolve conflicts on a more abstract or a more detailed level. Working out conflicts at a higher abstraction level saves time and coordination activities, but is more restrictive to agent interactions than it may be necessary. On the other hand resolving conflicts having more detailed forms of the plan enables finding more precise interaction solutions, but this can become a very complex search problem and so be ineffective. Whether it is profitable to enable quick but abstract coordination or to prefer slow but more accurate one, is mainly domain specific and should be carefully considered.

#### *Negotiation in Distributed Planning*

So far we discussed two approaches to distributed planning that aim at using coordination to enable conflict recognition and resolving. Coordination can be also

used to promote cooperative work. Coordination in multiagent systems can be realized in many different ways one of which is via negotiations between agents. But first let us define the term negotiation more precisely.

“Negotiation is a process by which a joint decision is reached by two or more agents, each trying to reach an individual goal or objective. The agents first communicate their positions, which might conflict, and then try to move towards agreement by making concessions or searching for alternatives

The major features of negotiation are (1) the language used by the participating agents , (2) the protocol followed by the agents as they negotiate , and (3) the decision process that each agent uses to determine its positions, concessions, and criteria for agreement.” (Huhns, Stephens, 1999, p. 104)

Researchers of negotiation mechanisms for coordination have separated into two different groups according to their approach to promoting productive and fair negotiation. The first group promotes negotiation using the environment as a negotiation media as well as a regulation mechanism. They tend to impose such rules on the environment that negotiation becomes (as described in Huhns, Stephens 1999):

- efficient – reaching an agreement by agents should be using minimum resources;
- stable – agents should not be allowed to stray/drawback from decisions that are already agreed on;
- simple – negotiation mechanisms have low computational and bandwidth demands on agents;
- distribution – there should be no central decision maker;
- symmetry – the mechanism of negotiation should be unbiased towards all agents.

As a result three types of environments have been identified: worth-oriented domains, state-oriented domains and task oriented domains. In this thesis we are interested in task-oriented domains (TOD) only. A definition of a TOD can be found in (Huhns, Stephens 1999, p. 105):

“A task-oriented domain is one where agents have a set of tasks to achieve, all resources needed to achieve the tasks are available, and the agents can achieve the tasks without help or interference from each other.”

An example of a TOD is the “Internet downloading domain” that we have introduced in section 2.1.2 of the current chapter. For the needs of negotiation the environment in this domain can be designed to impose the following rules and constraints:

1. as a first step all agents have to announce the documents they need;
2. documents that appear to be common for two or more agents are arbitrary assigned for downloading to an agent;
3. downloading of documents has a price;
4. each agent is provided with access to documents it has downloaded itself as well as to documents from its originally given set that are downloaded by other agents.

This set of rules guarantees that negotiation between agents will be simple, symmetric, distributed and efficient. Stability will depend on the agent strategies at the first step of the protocol. As it is best for an agent to declare the true set of document it needs (it can only benefit from being honest), no matter whether other agents are honest or not, there is no reason for agents to diverge from reached agreement and therefore the protocol is stable.

The second research group focuses on agent-centered negotiation mechanisms. Here the idea is to find out the best strategy for an agent in a particular environment. Two classes of approaches are developed so far – one of them is based on speech-acts and possible world semantics, and the other is based on the concept of economic rationality.

The idea of agents being economically rational was implemented by Rosenschein and Zlotkin in a unified negotiation protocol. It is based on the assumptions that the number of participating agents is limited and they share a common language and common problem abstraction, and they need to get to a common solution. Rosenschein and Zlotkin defined the notions of deal and utility for the needs of multiagent negotiation. Details on this approach can be found in Huhns, Stephens, 1999.

### ***2.3 Combining Planning and Execution***

In DAI planning is seldom a problem on its own, usually agents need to actually perform the planned actions in order to achieve their goals. Execution failures

and unexpected results of actions further complicate the coordination task agents face. In some cases the complete multiagent plan may be invalidated by some unexpected events or outcomes of actions.

Several methods have been developed to deal with the above problems. The easiest way to repair a multiagent plan is to repeat the planning and coordination of individual plans activities. This approach can be quite ineffective in uncertain domains and can cause significant delays at execution time. In order to avoid re-coordinating, agents can be designed to have access to a library of reusable plans when the original individual plan has failed. In other approaches such as the hierarchical behavior-space search repeating coordination can be avoided by resolving conflicts on abstract (higher) levels of plan representation. Using abstract plans when coordinating with others allows agents to replan in case of failure without affecting the multiagent plan.

Another approach uses the algorithm of plan merging but on larger plans. Those plans include branches of some section of the original plan that can be used as alternatives for achieving the same goal under different circumstances. So in execution time if an action fails or is inapplicable for some reason another branch of the plan can be used instead of initializing new planning and coordination processes. This method is known as **contingency planning** as the agent provides alternatives for contingencies at execution time. Although contingency planning deals with uncertainty of action execution to some extent, it is a lot slower and complicated at coordination time. Plans to be merged become quite large and the search space of possible conflicts becomes more difficult to build as all combinations of plan execution paths (following different branches) should be inspected. The latter problem can be simplified by associating branches with the conditions that show when a particular branch can be followed. Thus, combinations of branches with contradicting conditions are eliminated straight away.

Other approaches focus on enabling pre-planning coordination. Pre-planning coordination is based on the assumption that some restrictions can be considered by agents before they start planning. Some researchers attempt to impose **social laws** on agents. A social law can be defined as “a prohibition against particular choices of action in particular context” (Durfee, 1999, p.153). An example of social laws can be the driving laws such as the prohibition to enter an intersection on red light. In

multiagent systems social laws can be encoded by the designer of the system or can be derived

## ***2.4 Partial Global Planning***

**Partial Global Planning** (PGP) is an approach that "explicitly considers that an agent's plans will change over time due to outcomes of its actions, outcomes of others' actions, changes in the environment, changes to the agent's goals, or changes in the agent's perception of the multiagent context" (Durfee, 1996, p.235). In order to deal with the above problems it is assumed that plans are built using only agent's current and subsequently partial models of environment and other agents. Moreover these models are subject to continuous reformation in the course of action. There are several consequences for the planning in the PGP approach as stated by Durfee (Durfee, 1996, p.236):

"...planning must be interleaved with plan execution;

plans are inherently uncertain of achieving their effects;

plans should be optimally traded off in the interests of timely execution, lest the agent fails to take action because its model of the world changes faster than its formation of plans in response to a particular model of the world."

In the PGP approach the individual and multiagent plans are concurrently evolving and are mutually connected. When selecting the next action to perform, an agent chooses from the set of possible actions that results from its local plan construction. When the agent has to select between its potential local plans on the other hand, it has to consider the collective (multiagent) plans that define the courses of action for all agents that will satisfy global goals. Since there is no agent with a global view of the environment or a centralized controller to plan the actions of all other agents, collective plans have to be built from the agent's local plans, and vice versa, local plans must consist of actions applicable in the current situation, i.e. actions consistent with the collective goals and plans. Therefore, as the world state changes so will the agent's possible actions, its local plans and then its collective plans. As the collective plans change, so will the local plans and thus the possible actions. So in PGP there is "a continuous flow of information upward and downward at different levels of decision making" (Durfee, 1996, p.236).

As changes in the environment may happen asynchronously and it may take time until information about them reaches all the agents, it is assumed that no agent will ever have complete view of the situation. That is why the PGP approach allows for acting and planning simultaneously. Agents can construct their local plans, exchange information that helps them form their partial-global plans, then reform their local plans and choose actions accordingly. It is also possible to use some ready to use plans in response to certain events in order to speed up the planning process. If at some point an agent (or a group of agents) decides that the collective plan is no more appropriate in the current situation, it may take actions so that the plan is abandoned and some other collective procedure is followed.

Some research has been focused on reducing communications between agents that are updating each other about major changes in the environment. A general idea is to divide the planning space into localized parts where planning can be independently performed. This approach may significantly improve the computational speed and decrease the communication load but a knowledge engineer will have to determine how to divide the planning problem between agents. One possible division of a planning problem can be achieved by using several coordination relationships between tasks, such as: enabling, facilitating, inhibiting, etc.

PGP differs from all other earlier approaches to distributed planning, because it does not aim at generating a static sequence of actions that all agents have to strictly follow (and therefore is very vulnerable to unpredictable outcomes or changes in the environment). Instead it tries to provide the agents with the most recent models of dynamic changes in the environment.

## **2.4.1 Detailed Description**

### *Task decomposition*

An important prerequisite for the successful work/application of the PGP is that tasks have to be decomposable. It is obvious that agents might not necessary know what other agents do and what effects their actions may have on its own. As it is generally assumed that no single agent is aware of the global state or tasks, coordination is introduced to allow agents to get a certain degree of awareness while accomplishing their tasks.

### *Local Plan Formulation*

Prior to communicating with other agents in PGP an agent has to define its own goals and to determine what actions it should take in order to achieve them, i.e. it has to construct its local plan. Therefore, purely reactive agents cannot use PGP because they do not have an explicit representation of goals and action sequences. Another characteristic/specific of the local plans is that it is very uncertain and branched in order to deal with unpredictable results of previous actions and unforeseen changes in the environment.

### *Local Plan Abstraction*

While different courses of action help an agent to act in an unpredictable world they might be of no use and even redundant for the other agents. That is why plans are first abstracted and then communicated to other agents. According to Durfee "abstraction plays a key role in coordination that has to be both correct and computationally efficient" (Durfee, 1996, p.237). So in PGP agents are able to distinguish those important plan steps that can be of interest to the rest of the community.

### *Communication*

After abstracting their local plans agents have to communicate them so that everyone can create its models of joint activity. Then some questions arise as to what, when and to whom should be spread around. In PGP there is a meta-level organization (MLO) that accumulates the necessary knowledge to guide the communication process. MLO determines both the control structure and the information flow, i.e. it specifies who is interested in the plans of a certain agent and who can impose plans on agents based on a global view.

### *Identification of Global Goals*

The presumption that all tasks are inherently distributive allows the agents to recognize the goals of one or more agents as sub-goals of a common global goal. Since the agents might be aware only of portions of the global goal at a certain moment in time, we are speaking of *partial global goals*. Actually the problem agents have to solve, that of constructing a PGG, can be regarded as an interpretation problem - "we need to generate an overall interpretation (global goal) that explains the component data (local goals)" (Durfee, 1996, p.238). The knowledge needed to synthesize a PGG is an abstraction of the knowledge needed to integrate the results of

distributed tasks. As interpretations may be ambiguous so can local goals be considered as contributing to competing PGGs.

#### *Partial Global Plan Construction and Modification*

The next step after PGGs are identified is to integrate local plans contributing to same PGGs in a *partial global plan*. It will combine the concurrent abstracted activities of the corresponding agents. The agent constructing the partial global plan may analyze those concurrent activities to see whether coordination can be improved. In the PGP approach attention is paid to two means of improving coordination:

- facilitating other agents' goal/task achievement - by finishing related activities earlier, and
- avoiding redundant task achievement.

The search of possible actions to form a partial global plan uses a simple hill-climbing algorithm and evaluation functions that considers characteristics such as:

- is it likely that task to have been achieved already by another agent;
- what time is it likely to take;
- can its results be useful for other agents.

The general *algorithm for reordering of the plan steps* (Durfee, 1996, p.238) is:

1. For the current order of actions, evaluate each action and sum up the results.
2. For each action, find the most highly rated of the agent's actions that are ordered after the current one. If it is higher rated, swap the actions.
3. If the new ordering is more highly rated than the current one, then replace the current with the new one and go to step 2.
4. Return the current ordering.

#### *Planning for Communication*

When an agent has formed a partial global plan it should next consider what information should be communicated to other agents. "In PGP, interactions, in the form of communicating the results of tasks, are also planned" (Durfee, 1996, p.238). Using the partial global plan an agent can decide which actions performed by one agent may be of interest to another and then it explicitly plans the transfer of their results. If an agent realizes that some results have to be synthesized it will use PGP to

construct a tree of exchanges such that partially synthesized result will be gathered by the same agent which can then build up the result.

#### *Acting after a partial global plan*

After the partial global plan has been constructed and communication has been planned the next step is to follow it, i.e. to translate back to the agent's local level what actions it should perform. For that reason the agent modifies its local plans to reflect all changes to its partial global plans. These modified local plans are then used by the agent to select next action.

#### *Ongoing Modifications*

During the process of planning and executing actions some changes in the environment may happen. This may cause a change of tasks or of actions achieving these tasks. An important question in that case is "when the changes to local plans are significant enough to warrant communication and re-coordination" (Durfee, 1996, p.239). The agents should be neither too sensitive to changes, transmitting even the small ones to one another causing communication overload, nor too insensitive, which will cause uncertainty as each agent expects the others to act after their partial global plan and it is no more followed closely. To deal with these problems in PGP the designer can set a parameter that defines the threshold of deviation from planned activity.

#### *Task Reallocation*

The process of task decomposition does not guarantee a spatial distribution of tasks among agents. Using their partial global plans, agents can determine whether they are overloaded with tasks or hardly engaged. They can also find underburdened agents and then generate and communicate new partial global plans that represent some agents taking some of the tasks. A recipient could suggest another partial global plan corresponding to its idea. In that way a kind of contracting procedure is initiated between the agents. If the negotiations are successful, tasks will be reallocated.

#### *Pros and Cons*

The PGP is the first approach that implements coordination both for task sharing and for result sharing. Another advantage is that planning is continuously performed in an incremental fashion, which makes it appropriate for evolving and distributed task domains. Nevertheless, there are a few limitations of this approach. Durfee outlines three of them (Durfee, 1996, p.243), namely the following:

- there is an implicit assumption that it is acceptable to proceed with actions even if there is no global plan yet. While this is an advantage in dynamic domains it caused problems in the air traffic control testbed (and so in other physical domains). To deal with that problem PGP has been extended by adding utilities to actions so that risky actions until their results are predicted with sufficient certainty;
- the algorithms used and relationships selected (mainly coordination relationships) are domain specific - for the vehicle monitoring task. Efforts have been made in the next years to generalize both algorithms and relationships.
- the coordination principles implemented require certain level of abstraction of the agent activities. This may cause computational problems when used for a society of tens of agents, as each PGP will have to consider the actions of every agent.

## ***2.5 Reactive Approach to Distributed Planning***

There are several architectures of reactive agents: the subsumption architecture developed by Brooks, the task competition approach defined by Ferber and the situated action theory proposed by Suchman, are some of them. Reactive agents have proved to be very appropriate for simulations, robotics and problem solving tasks.

The simplest and earliest architecture of reactive agents was based on **situated rules** and was proposed by Wavish and Cohen. There, reactions to situations were described by a number of simple rules in the following format:

*if <perceived situation>*

*then <specific actions>*

While these rules are simple and convenient for describing simple behaviors, they are inapplicable in complex situations. This is due to several suppositions on which situation rules are based:

- there is no memory of past events or states,
- actions come only as responses to perception,
- rules should be mutually exclusive and not conflicting to each other - this alone is a very restrictive hypothesis.

In the next years Brooks has proposed a new architecture, called the **subsumption architecture**, which overcame the limitations of situated rules. It used more elaborate descriptions of behavior for robots with different abilities. The subsumption architecture actually consists of a hierarchical set of augmented finite state automata that are linked via a master-slave inhibition relationship. Each automaton module of a higher level can inhibit the flow of information into lower level modules or commands sent for execution. Modules are also grouped into layers so that each layer implements a different knowledge domain of the agent. Unfortunately, even with this approach there are some complex behaviors (such as conflicts between tendencies at the same level) which are hardly possible to implement using simple inhibitory techniques.

Another architecture is that of **competing tasks** in which the behavior of agents is defined by competition between tasks. Only one task can be performed at a time and it will take control of the agent's actuators. Here a task is defined as "high level behavioral sequence, opposed to low level actions performed directly by actuators" (Ferber, 1996, p.292), and considered as primitives. Tasks are selected and suspended using perceptions and feedback both from the environment and from agent itself. Reinforcement learning is also used to make the agent more sensitive to stimuli that have already triggered a task. Thus agents are more effective when repeating a task and are avert to performing new ones.

Neural networks can be used to produce more uniform behavior. Generally simple networks are unable to produce complex behaviors. To make networks adaptable to agent environment and tasks we can use genetic algorithms as an optimizing function. The main advantage of neural networks over other reactive architectures is the fact that previous actions effect future behaviors.

There are two very important external factors for the organization in the system of reactive agents: the **structure of the environment** and the **feedback mechanisms**.

The **structure of the environment** is the basis on which the agent society structure is built. In that process organizational structures are formed in correspondence to agents' spatial locations. Agents in different regions of the world will gather in specialized groups and will have different social roles. Their physical location also defines agents' behavior because stimuli are spread as a reciprocal function of the distance to the agent.

Different **feedback mechanisms** are also used to induce stable states and trends of behavior. Positive feedback makes agents specialize in typical behaviors while negative feedback results in conservation of the current social structure. Ferber distinguishes two categories of feedback (Ferber, 1996, p.290):

- *local feedback* - which is built-in in the agents architecture by the system designer;
- *global feedback* - that comes as a result of interactions between agents and whose effect is not explicitly encoded at agent level.

An example of positive feedback may be a reinforcement mechanism that inclines an agent to perform tasks it is already specialized in. This feedback mechanism can be implemented at design time on the agent level and is therefore a local one. Regulations of role distribution, on the other hand, can be viewed as a negative feedback that is a result from agent interactions and hence is a global one.

## ***Chapter 3 Planning Research Library***

### ***3.1 Purpose and Some Technical Aspects of the System***

The **Planning Research Library** is designed as a result of the research presented in chapter one and two of this thesis. The purpose of the system is to provide an object model of agents, agent systems and planning methods. That library can be used for planning simulations by students and researchers in the field of DAI. It can help its users (we will refer to them as users although they need to do some coding) choose a planning method to implement in an agent system they have to develop. The **Planning Research Library** can also be used to illustrate some common planning methods in MAS courses.

The design of the system is made using Together 4.2 and code implementations are made in Java language. The system is compilable with jdk 1.3.0. The project classes are grouped in four packages. The class diagram for each package can be found in Appendix A and the complete Java source in Appendix B.

### ***3.2 Modeling Agents***

The models of types of agents as well as their related (and embedded) concepts are defined in the agent package of the **Research Planning System**.

#### *Agent and Derived Classes*

The Agent class is representing the Ferber's definition of an agent (see section 1.1.1) – an agent is existing in an environment, possesses some capabilities it can utilize when achieving some goals, it has some internal resources and can perceive its environment. Agents are also capable of: acting in the environment, sensing that environment and reproducing themselves in it. They are also capable of manipulating with environmental objects so that to create or destroy them, perceive and modify them in certain ways. Each agent is designed to operate in a different thread in order to guarantee asynchronous computation.

Some optional characteristics (such as mobility and communicability) of agents are modeled by Java interfaces. Those interfaces specify the methods the agent needs to implement when having corresponding characteristic. For example, if we

want an agent to be communicating with others, it needs to be able to talk and to listen to other agents (as specified by the `CommunicatingAgent` interface).

In accordance with section 1.2.2 agents are separated into two main types with respect to their architecture – `ReactiveAgent` and `CognitiveAgent` classes. A `ReactiveAgent` usually does `reactToStimuli()` while a `CognitiveAgent` is supporting an environment model and is capable of tracking changes in the environment caused by other agents. The behavior reactive agents exhibit is initiated by a set of rules, while cognitive agents are capable of creating plans and willingly performing an action (in order to follow plans). Cognitive agents are also capable of saving created plans for future usage. Those two classes of agents are further extended to classes that model agents with architectures that are developed for certain planning methods. Currently there are `SituatedAgent`, `PGPAgent` and `TaskDrivenAgent` classes.

`TaskDrivenAgent` is designed to perform task-driven planning and to enhance task sharing and result sharing mechanisms (that have been described in section 2.1.3). Hence, a task driven agent can allocate tasks to as well as receive tasks from other agents. So it needs to store references both to agents that will achieve tasks for it and to agents it is working for. In the `TaskDrivenAgent` class two different types of containers are used to store that information. When the agent is `isOverburdened()` it can decompose its current task and reallocate subtasks to free agents. A `TreeMap` structure is used to keep the references to agents that will perform different tasks from the decomposition tree. When the agent has to `composeSolution()` it refers to the task decomposition tree map to find out how to combine results from different agents. On the other hand, a `HashMap` is saving couples with the task as a key and the contractor (the agent that has reallocated the task to this agent) agent as a value. When the task driven agent finishes working on a task it retrieves the agent to whom to send results from that `HashMap` structure. Then it can `propagateResults()` back to the contractor agent while at the same time it can announce those results to other potentially interested agents (thus the same method is used to enhance result sharing).

Task and environment specific methods are not implemented in the `TaskDrivenAgent` class. Their bodies contain no operators but are compilable so that if some method is of no use in a particular domain it need not be encoded.

It is assumed for the purposes of this thesis that the accomplishment of a task by a `TaskDrivenAgent` is limited to generating a plan. Problems of task execution are not regarded here.

A PGPAgent is a model of an agent that acts after the Partial Global Planning method (described in section 2.3). This method is also aiming at task-driven planning and hence PGPAgent inherits TaskDrivenAgent. As communication is basic to this planning approach, the PGPAgent is designed to be implementation of CommunicatingAgent. To enhance the PGPlanningMethod the agent is supporting the following methods: abstractLocalPlan(), formulateLocalPlan(), identifyGlobalGoals(), etc.

SituatedAgent is the simplest architecture for reactive planning – situated rules (see section 2.4.1 for description). Rules can be added, removed and modified in the agent's storage. The agent is acting after some rule when perceiving that its conditions are satisfied.

#### *Related Concepts*

The Plan class provides a most common definition of plans. A plan is consisting of the goals it has to achieve and the sequence of actions that will achieve them. All kind of modification methods for managing goals and action sequence are provided (such as add and remove methods). In the implementation of the Plan class, plans are regarded as linear sequences of actions. This total order of the plan is always needed for execution purposes since it is assumed that an agent can execute one action at a time. For more sophisticated purposes, such as task sharing, plans may have to be stored as partial order structures (graph). It is up to the user of the system to define the most suitable form of plans for her/his planning purpose and implement it in a derived class. As an addition to the other methods there is a an add(Plan) method that simply concatenates two plans without merging them and checking for conflicts.

#### *Planner and ClassicPlanner*

Plans are product of the work of an agent's internal component called Planner. When creating a plan, the planner is aware of the plan goals, the state of the world to which the plan should be applied, and the set of actions that can be used for accomplishing the goals. The planner is also having internal representations of the evolving plan and the protected goals. As the Planner class is too general it does not provide definitions of its functional methods (they are declared abstract). The user of the system has to implement the specific algorithms a particular planner will use for goal ordering, goal protection, action selection, resolving conflicts and goals reordering.

A more specific model of a planner is provided by the `ClassicPlanner` class (that extends `Planner`). Its implementation is based on the specific algorithms and assumptions of a single agent planner presented in section 1.3.

The `checkForConflicts()` method of the `Planner` class is declared as abstract so that the user should define at what point of the planning process and in what way the emerging plan will be checked for conflicts. Hence, in the `ClassicPlanner` class this method is implemented so that it is checked for conflicts before an action is added to the plan. Thus, the plan is guaranteed to be coherent and consistent at every point of its generation.

The `Planner` class contains a pool storing all protected goals. This pool is further detailed in the `ClassicPlanner` class to contain a collection of `Literal` objects, since goals in the `ClassicPlanner` representation are conjunction of literals. Protecting a goal of `ClassicPlanner` is realized as protecting each of the literals that form this goal as a conjunction. In that way no action contradicting any of the conjunctions of a goal (not only the goal as a whole) can be performed until the goal is unprotected.

`ClassicPlanner` is using `STRIPSAction` (defined in `utils` package) representation, where an action is defined as preconditions and postconditions. An action `isApplicable()` if its preconditions are satisfied in the current state of the world. Action selection in `ClassicPlanner` is based on using some heuristic methods such as the number of non-linkable preconditions, the number of bonus goals the action can accomplish and the resources it will consume.

Each agent that has to create plans not only for itself but other agents has to implement `MultiagentPlanner`. For example, `TaskDrivenAgent` is utilizing `MultiagentPlanner` to allow each agent in a system to be able to plan for all. `MultiagentPlanner` enables not only centralized planning for multiple agents but also plan coordination and synchronization. The user has to implement those methods for the particular planning method to be used.

### ***3.3 Modeling Systems of Agents***

Concepts related to modeling of systems of agents are grouped in the `agentsystem` package of the project. It contains a class hierarchy defining types of agent systems as well as different multiagent planning methods that are described in section 3.4.

An AgentSystem is generally defined by its name, type and the agent population it hosts. Our implementation of an agent system is based on the Ferber's definition provided in section 1.2.1. Thus the AgentSystem stores information of the environment in which agents operate as well as the relations that connect them and the operations the agents can perform in the environment. The system of agents also has to be aware of the multiagent planning method that will be used when forming the multiagent plan. An AgentSystem usually provides means for managing its agent population, relation and operation sets. With regard to the planning purpose of the modeled AgentSystem, a method allowing the agents to follow the generated multiagent plan is defined. As different algorithms for plan execution can be used in different systems this method should be implemented by the user in a derived class.

As it is explained in section 1.2.1 two types of agent organization are distinguished – MultiagentSystem and DistributedSystem structures. As a multiagent system is an open organization hosting different kinds of agents each pursuing its own goals, the model of a MultiagentSystem offers means for registering and unregistering agents in the society. It also provides a default implementation of the plan following method that specifies how agents coordinate to follow the common multiagent plan.

A DistributedSystem on the other hand assumes that its designer is specifying the number and type (one type of all agents) of the agents that solve a common task, together with the resource, skills and task (subtask) distribution among agents. Therefore, methods for managing all of these containers are supported together with a general implementation of the plan following method. Task relationships (as defined in the utils package) are stored to provide means for decreasing communication (especially with result sharing algorithms).

A MultiagentPlan is modeled as a storage saving the plans for each of the agent it encompasses. Methods for managing and accessing that storage are provided.

### ***3.4 Implementing Multiagent Planning Methods***

As we have explained above the differences between distributed and multiagent systems are reflected in the corresponding classes. While in a distributed system all agents are created at design time, multiagent systems are open societies where agents share a common environment while pursuing tasks of their own. For the purposes of this thesis it is assumed that methods for task driven planning are

specified for distributed systems, and methods for plan coordination are proper for multiagent systems. Of course this is a rigid separation and also not all agent systems can be strictly divided into those two classes. Some hybrid systems can exist that are using different planning methods but they need to be separately modeled by the user.

The **Planning Research Library** models all multiagent planning methods that are introduced in chapter 2 of this thesis. A general definition of a multiagent planning method is provided by the `PlanningMethod` class.

A `PlanningMethod` is identified by its type and name and is characterized by a number of planning agents that `generatePlan()` for the agents participating in the particular `AgentSystem` (or at least for the ones the method is aware of). The `PlanningMethod` defines the way in which to `selectPlanningAgents()`.

The `CPDPPlanningMethod` class implements mechanisms for centralized planning for distributed plans. It provides means for choosing the single agent that will plan for all the rest. That agent can be either predefined at design time or selected (via negotiation or other mechanism) among the participating agents. As this approach is proper for distributed systems it is assumed that participating agents represent the `TaskDrivenAgent` class. The planning agent then utilizes the `MultiagentPlanner` component it possesses to `generatePlan()` for the multiagent system.

The mechanisms for distributed problem solving that are implemented in `TaskDrivenAgent` together with task relationships defined in `utils` package can be used to enhance Distributed Planning for Centralized Plans. That is why no separate `PlanningMethod` class is designed.

The planning method of centralized plan merging is implemented as a representative of plan coordination approaches to multiagent planning. The `CentralizedPlanMergingMethod` is characterized by a single agent that gathers and merges the plans of all other agents in the system to form the multiagent plan. Plan merging is implemented as putting synchronization actions in individual agent plans.

The `PGPlanningMethod` is defined after the method presentation in section 2.3 of this thesis. It maintains a `PGPlan` containing the abstracted activities of the registered agents. Agents using this approach are assumed to be `PGAgents` that are capable of abstracting their local plans and of identifying of partial global goals that are stored in a special container. This method also allows agents to plan their communication activities and to modify their local plans as well as the `PGPlan` during execution.

### ***3.5 Modeling Environment***

The concept of environment in multiagent systems is a generalized form of the concept of world in classic domains. That is the reason why we will use the term world as a synonym of environment, which is particularly useful when tasks are modeled because their definitions usually define a world not an environment.

An Environment generally has a volume that we have implemented as dimensions property. Unlike Ferber's definition of multiagent system (see section 1.2.1) in **Planning Research Library** objects that are part of the world are modeled as attributes of the environment. This is reasoned by the fact that usually world objects are described as part of the world when a problem is formulated. World objects can be updated, manipulated and managed (removed or added) by agents. Our implementation of Environment also stores and provides access to a property modelling the current state of the world. When an agent attempts to perform an action the `changeWorldState(Action)` method of Environment is used to simulate the impacts this action will have on the world.

Optional characteristics of the environment such as the ones defined in section 1.1.2 are modeled by particular Java interfaces. Each interface contains methods that ensure the modeled characteristic. For example, if an environment is implemented as deterministic it has to provide only one result to each action performed in a certain world state. On the other hand, implementing `NondeterministicEnvironment` includes providing means for getting all possible results of an action. The complete list of environment interfaces can be found in Appendices A and B.

Environment class uses a general definition of `WorldState` including a time model, and containers for storing the relations that hold in the current world state and the stimuli it broadcasts to agents. A more specific representation of a world state is provided in `STRIPSWorldState` class. It is modeling a STRIPS world where states are defined as conjunctions of `Literal` objects. Changing a world state is implemented as removing and adding certain literals to the current set.

### ***3.6 Auxiliary Concepts Modeled***

All of the auxiliary and common concepts that have to be modeled for different packages of the **Planning Research Library** are placed in the `utils` package.

Here we will briefly present the implementations of the basic concepts in this package.

### *Task*

The Task class is designed to model tasks that agents have to accomplish. Since tasks can be very different in their nature we have left the implementation of particular tasks up to the user of the system. Derived classes of the Task class have to be specifically coded for each domain and should be in relation to agent capabilities. Only after designing and implementing the tasks agents have to deal with in a particular system of agents the user can experiment with different planning methods in this particular domain.

Both problems and subproblems (created as decompositions of the same problem) are modeled using the Task class. Thus, recursive algorithms for task decomposition are easily coded.

In our generalized model tasks are defined as having a goal that should be accomplished in the given environment. That task will possibly include managing objects some of which are static and some of which are moveable. The task goal has to be achieved by applying some of the specified possible actions in the provided initial state of the world.

The solution of the problem and the process of reaching it have to comply with certain requirements that are saved as part of the task. So far requirements are modeled as posing time and resource limitations. Other requirements (solution requirements for example) can only be saved as comments and can be used for reference purposes. In a future (or a user defined) implementation of the Requirement class some content language can be used for representing other limitations.

### *Goal*

The Goal class provides a very generic form of a goal. It allows goals to be protected and to be put in relations to each other. More specific implementation of a goal can be found in the ConjunctiveGoal class. There a goal is represented as a conjunction of literals that have to be true in a certain world state to make the goal achieved. Goal protection over conjunctive goals is implemented as protection of every single literal comprising the goal.

### *Action*

An Action is generally defined by its name, and the resources that will be consumed when executed. The details of action execution and applicability to a

certain world state are action specific and should be implemented when modeling a particular action. A more detailed definition of action is presented in the STRIPSAction class. A STRIPS operator (action) is characterized by its preconditions and postconditions that in this presentation are assumed to be conjunctions of literals. An action is applicable when its preconditions are satisfied in the current world state. Action execution is realized by applying its postconditions to the current world state to produce a new world state. Macro actions are also defined to allow planners work more efficiently by using stored sequences of actions.

### ***3.7 User Interface Specification***

On the basis of the developed **Plan Research System** a graphic user interface can be defined to allow users easily create multiagent systems, choose tasks, add agents and then monitor the process of plan creation and coordination. A typical user scenario of this user interface system can be:

1. Create an agent system of a predefined type and create the environment in which tasks will be achieved.
2. Create agent (instance of an implemented class) of a certain type that possesses some skills and can offer some services.
3. Multiply the already created agent and add it to the agent system.
4. Choose a planning task to experiment with, planning tasks are only predefined they cannot be created by the system.
5. Set relationships between agents and between agents and objects.
6. Choose a multiagent planning method.
7. Start the planning process and monitor it. Compare the results of the planning process in terms of time for plan creation and plan characteristics.

This software system is limited to creating and viewing multiagent plans. It does not address issues regarding the plan execution.

## *Chapter 4 Conclusions and Future Work*

### *Conclusions*

Multiagent systems are a rapidly developing research area in the field of AI. Currently there is no consensus on definitions of the most basic concepts of MAS – agent and system of agents. Different types of agents and multiagent systems use different planning methods that have been developed to suit the specifics of each task and agent system. That makes the selection of a particular planning method to implement in a newly modeled MAS a complicated task requiring a lot of research and testing until the right method is found. This MSc thesis provides a source of comprehensive information about the most popular multiagent planning methods. On that base an object model (**Planning Research Library**) is designed to enable researchers and students in the field of MAS to experiment with different agent architectures, types of agent systems and planning methods before choosing the right ones to be implemented for a particular domain and set of tasks. This library can be also used to illustrate (for example at lectures on MAS) the most popular planning and coordination techniques work in different systems of agents. It is designed to be easily extensible so other applications can be found in the future.

### *Future Work*

The **Planning Research Library** we have presented in this thesis can be further developed and improved in four main directions:

- it can be extended – more classes can be implemented to model other multiagent planning methods and algorithms, other tasks to be solved and other agent architectures to be experimented with;
- some work can be done to make it more user-friendly and error-proof – for example, exceptions can be defined to handle unexpected situations. Currently, there are many inherent assumptions coded in the system that the designer has to consider. This can be overcome with assertions and other verification mechanisms that can be implemented in basic classes;
- user-interface can be coded and further developed to offer all of the defined basic features as well as new ones. Some classes can be

defined to allow users to customize and easily extend the system interface;

- as another option user interface can be implemented using JSP. In that way the **Planning Research Library** can be accessed over the Internet and it can be made open to many users that could work at the same time.

## *References*

1. Durfee E. (1999) Distributed Problem Solving and Planning, In Multiagent Systems – A modern approach to Distributed Artificial Intelligence, Gerhard Weiss (ed.). MIT Press.
2. Ferber J. (1999) Multi-Agent Systems – An introduction to Distributed Artificial Intelligence. Addison Wesley Longman.
3. Kiss G. (1996), Agent Dynamics, In Foundations of Distributed Artificial Intelligence, G. O’Hare and N.Jennings (ed.). John Wiley & Sons, Inc.
4. Huhns M., Stephens L. (1999), Multiagent Systems and Societies of Agents In Multiagent Systems– A modern approach to Distributed Artificial Intelligence, Gerhard Weiss (ed.). MIT Press.
5. Hendler J. (1999) Making Sense out of Agents, In IEEE Intelligent Systems, March/April issue, pp 32-37
6. Wooldridge M, Jennings N. (1995) Intelligent agents: Theory and practice, In The Knowledge Engineering Review, vol. 10(2), pp115-152.
7. Hermans B.(1999), Intelligent Software Agents on the Internet, URL:
8. Wooldridge M. (1999) Intelligent Agents, In Multiagent Systems – A modern approach to Distributed Artificial Intelligence, Gerhard Weiss (ed.). MIT Press.
9. Moulin B., Chaib-Draa B. (1996), An Overview of Distributed Artificial Intelligence, In Foundations of Distributed Artificial Intelligence, G. O’Hare and N.Jennings (ed.). John Wiley & Sons, Inc.
10. Weiss G. (1999), Multiagent Systems– A modern approach to Distributed Artificial Intelligence, Gerhard Weiss (ed.). MIT Press.
11. Rosenschein J. (1993), Consenting Agents: Negotiations mechanisms for multiagent systems, In Proc. Int. Jt. Conf. Artif. Intell., 13 th, Chambery, France, pp – 792-799
12. Van Dyke Parunak H. (1996), Applications of Distributed Artificial Intelligence in Industry, In Foundations of Distributed Artificial Intelligence, G. O’Hare and N.Jennings (ed.). John Wiley & Sons, Inc.
13. Vere S. (1992) Planning, In Encyclopedia of AI, Shapiro (ed.), Second Edition

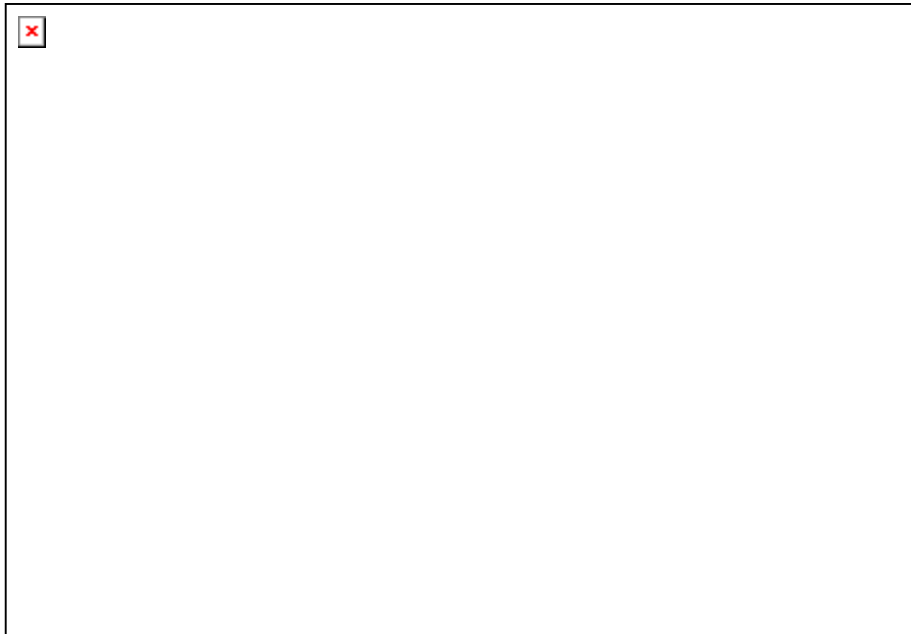
14. Durfee E. (1996), Planning in Distributed Artificial Intelligence, In Foundations of Distributed Artificial Intelligence, G. O'Hare and N.Jennings (ed.). John Wiley & Sons, Inc.

15. Ferber J. (1996), Reactive Distributed Artificial Intelligence: Principles and Applications, In Foundations of Distributed Artificial Intelligence, G. O'Hare and N.Jennings (ed.). John Wiley & Sons, Inc.

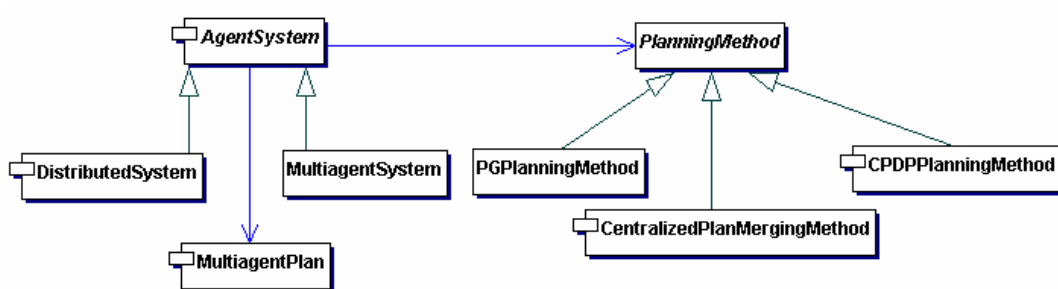
# Appendix A: Planning Research Library

## Class Diagrams

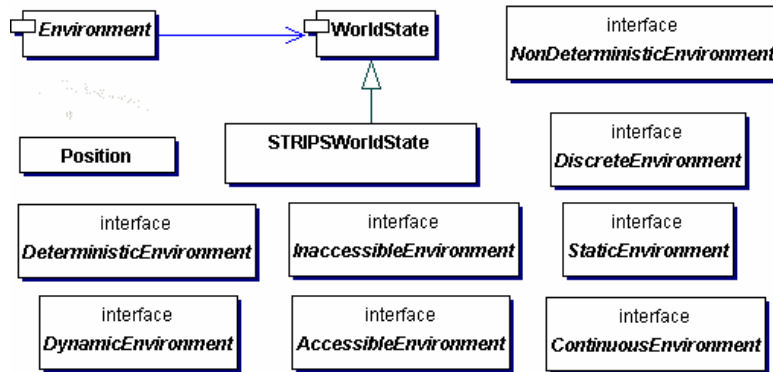
Agent Package



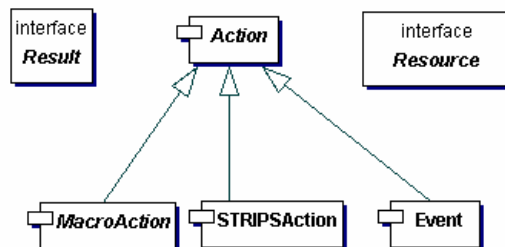
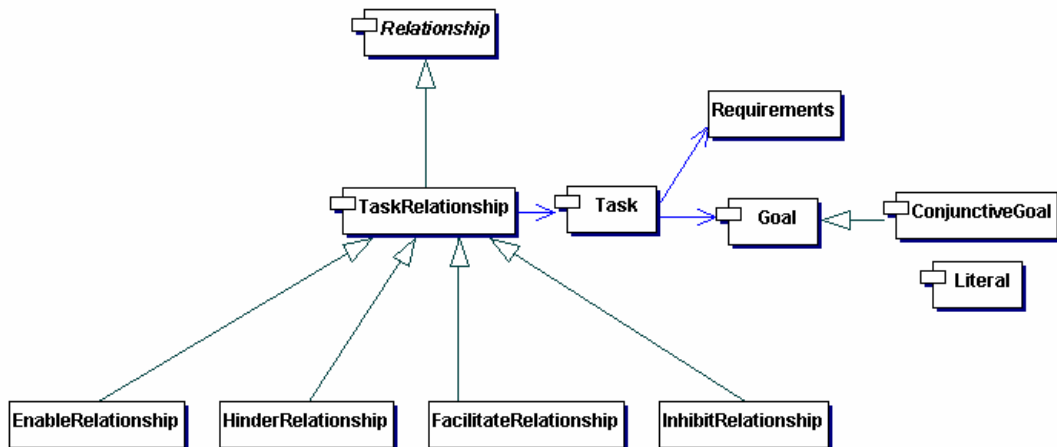
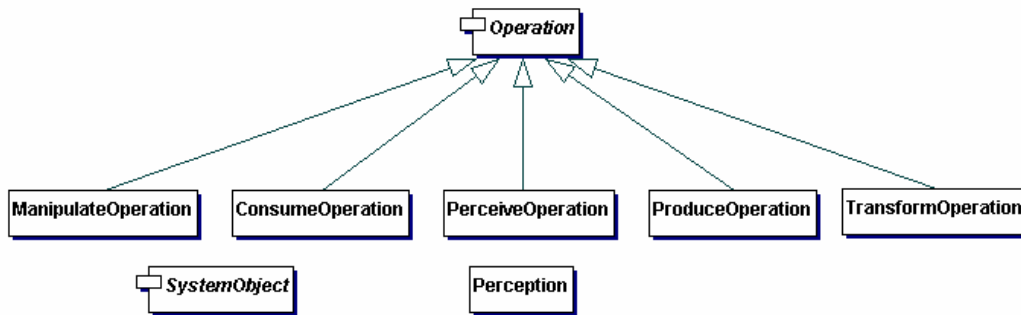
Agent System Package



## Environment Package



## Utils Package



# *Appendix B: Planning Research Library*

## *Source Code*

### *Agent Package*

```
//Agent Class definition
package agent;
import java.util.Set;
import java.util.Vector;
import utils.*;
import environment.*;

public abstract class Agent {
    private Perception m_agentPerception;
    // a Set of Action objects
    private Vector m_agentCapabilities;
    private Vector m_agentGoals;
    // Vector of resources available to the agent
    private Vector m_resources;
    // Vector of services that the agent offers
    private Vector m_services;
    private Environment m_environment;

    public Agent() {
        m_agentPerception = new Perception();
        m_agentCapabilities = new Vector();
        m_agentGoals = new Vector();
        m_resources = new Vector();
        m_services = new Vector();
    }

    //accessor methods
    public Vector getGoals() {
        return m_agentGoals;
    }
    public void setGoals(Vector goals) {
        m_agentGoals = new Vector(goals);
    }
    public Perception getPerception() {
        return m_agentPerception;
    }
    public void setPerception(Perception agentPerception) {
        m_agentPerception = agentPerception;
    }
    public Vector getAgentCapabilities() {
        return m_agentCapabilities;
    }
    public void setAgentCapabilities(Vector agentCapabilities) {
        m_agentCapabilities = new Vector(agentCapabilities);
    }
    public Vector getResources() {
        return m_resources;
    }
    public void setResources(Vector resources) {
        m_resources = resources;
    }
    public Vector getServices() {
        return m_services;
    }
}
```

```

    }
    public void setServices(Vector services) {
        m_services = services;
    }

    public boolean addGoal (Goal newGoal) {
        return m_agentGoals.add(newGoal);
    }
    public boolean removeGoal (Goal goal) {
        return m_agentGoals.remove(goal);
    }
    public boolean addCapability (Action capability) {
        return m_agentCapabilities.add(capability);
    }
    public boolean removeCapability (Action capability) {
        return m_agentCapabilities.remove(capability);
    }

    //virtual methods
    public abstract void exist();
    public abstract void act();
    public abstract boolean reproduce();
    public abstract void senseEnvironment();
    public abstract boolean performAction(Action action);

    public abstract boolean createObject();
    public abstract boolean destroyObject();
    public abstract boolean perceiveObject();
    public abstract boolean modifyObject();
}
// end of Agent Class definition

// ClassicPlanner Class definition

package agent;

import java.util.Vector;
import java.util.Stack;
import utils.*;

public class ClassicPlanner extends Planner {
    //Vector of Literal objects that are protected
    private Vector m_protectedLiterals;

    public Vector getProtectedLiterals() {
        return m_protectedLiterals;
    }
    public void setProtectedLiterals(Vector literals) {
        m_protectedLiterals = literals;
    }
    public void addProtectedLiteral(Literal literal) {
        m_protectedLiterals.add(literal);
    }
    public void removeProtectedLiteral(Literal literal) {
        m_protectedLiterals.remove(literal);
    }
}

    public void protectGoal(Goal goal) {
        goal.setGoalProtection(true);
    }

```

```

        if (goal instanceof ConjunctiveGoal) {
            m_protectedLiterals.add(((ConjunctiveGoal)goal).getLiterals());
        }
    }

    public void unprotectGoal(Goal goal) {}

    public Plan createPlan(Goal goal) {
        Plan linkedPlan = new Plan();
        if (linkNode(goal, linkedPlan)) {
            return linkedPlan;
        } else {
            return expandNode(goal);
        }
    }

    public void addAction(Action action) {
        getCurrentPlan().addAction(action);
    }

    //to implement
    public void reorderGoals() {}
    public Plan expandNode(Goal goal) {return new Plan();}
    public boolean linkNode(Goal goal, Plan outPlan) {return true;}
    public boolean isPlanningComplete() {return true;}
    public boolean checkForConflicts(Action action) {return true;}
    public void resolveConflicts() {}
    public Action selectAction() {return new STRIPSAction();}
    public boolean isLinkable(Literal literal) {return true;}

    //next methods provide some criteria for action selection
    public int getNumOfNonlinkablePreconditions(Action action) {
        Vector preconditions = ((STRIPSAction)action).getPreconditions();
        int numOfPreconditions = preconditions.size();
        int nNonLinkableNum = 0;
        for (int i = 0; i < numOfPreconditions; i++) {
            Literal literal = (Literal)preconditions.elementAt(i);
            if (!isLinkable(literal)) {
                nNonLinkableNum++;
            }
        }
        return nNonLinkableNum;
    }

    public int getNumOfBonusGoals(Action action) {
        Vector postConditions = ((STRIPSAction)action).getPostconditions();
        int numOfPostConditions = postConditions.size();
        int nBonusGoalsNum = 0;
        Stack goals = getPlanGoals();
        for (int i = 0; i < numOfPostConditions; i++) {
            if (goals.search(postConditions.elementAt(i)) != -1) {
                nBonusGoalsNum++;
            }
        }
        return nBonusGoalsNum;
    }

    //Returns a vector of Resource objects that need to be consumed
    //for action completion
    public Vector getAmountOfResources(Action action) {
        return action.getResources();
    }
}

```

```

    public Action allowUserToSelect() {return new STRIPSAction();}
}
// end of ClassicPlanner Class definition

//CognitiveAgent Class definition

package agent;
import java.util.HashMap;
import java.util.Vector;
import environment.*;
import utils.*;

public abstract class CognitiveAgent extends Agent {
    private Planner m_planner;
    private Vector m_otherAgents;
    //a Set of ready to use plans
    private HashMap m_routinePlans;
    private Plan m_plan;
    private EnvironmentModel m_environmentModel;
    // contains the tasks that the agent has engaged to complete
    private Task m_task;

    public CognitiveAgent() {
        m_planner = new ClassicPlanner();
        m_otherAgents = new Vector();
        m_routinePlans = new HashMap();
        m_plan = new Plan();
        m_environmentModel = new EnvironmentModel();
        m_task = new Task();
    }
    public CognitiveAgent( Planner planner, HashMap routinePlans) {
        this();
        m_planner = planner;
        m_routinePlans = routinePlans;
    }

    //accessor methods
    public EnvironmentModel getEnvironmentModel() {
        return m_environmentModel;
    }
    public void setEnvironmentModel(EnvironmentModel environment) {
        m_environmentModel = environment;
    }
    public Task getTask() {
        return m_task;
    }
    public void setTask(Task task) {
        m_task = task;
    }
    public Planner getPlanner() {
        return m_planner;
    }
    public void setPlanner(Planner planner) {
        m_planner = planner;
    }
    public Vector getOtherAgents() {
        return m_otherAgents;
    }
}

```

```

public void setOtherAgents(Vector agents) {
    m_otherAgents = new Vector(agents);
}
public void setRoutinePlans(HashMap plans) {
    m_routinePlans = new HashMap(plans);
}
public HashMap getRoutinePlans() {
    return m_routinePlans;
}
public Plan getPlan() {
    return m_plan;
}
public void setPlan(Plan plan) {
    m_plan = plan;
}

public boolean executePlan(Plan plan, Vector outResults) {
    return true;
} //vector outResults will contain the results of plan execution
public void senseEnvironment() {}
public boolean performAction(Action action) {
    return true;
}
//can be overridden for task updates
public void updateTask() {}

public abstract boolean accomplishTask(Task task);
public abstract boolean updateEnvironmentalModel();

}
// end of CognitiveAgent Class definition

```

```

// CommunicatingAgent interface definition

```

```

package agent;
import java.util.Vector;
public interface CommunicatingAgent {
    public void talkToOthers(Vector agents);
    public void listenToOthers();
}
// end of CommunicatingAgent interface definition

```

```

// EnvironmentModel Class definition

```

```

package agent;

public class EnvironmentModel {
}
// end of EnvironmentModel Class definition

```

```

// MobileAgent interface definition

```

```

package agent;

import environment.*;

```

```

public interface MobileAgent {
    //selfpropelling
    public boolean move(Position position);
    public boolean stop();

    //manipulations with objects
    public boolean moveObject(Object obj, Position pos);
    public boolean catchObject(Object obj);
    public boolean takeObject(Object obj);
    public boolean putObject(Object obj);
}
// MobileAgent interface definition

// MultiagentPlanner Class definition

package agent;
import agentsystem.*;
public abstract class MultiagentPlanner extends Planner {
    public void synchronizeAgentsPlans(){}
    public MultiagentPlan createMultiagentPlan() {return new MultiagentPlan(); }
}
// end of MultiagentPlanner Class definition

// PGPAgent Class definition

package agent;

import utils.*;

public class PGPAgent extends TaskDrivenAgent implements CommunicatingAgent{
    private Plan m_pgPlan;
    /*
    public Plan abstractLocalPlan() {}
    public Plan constructPGP() {}
    public void communicateToOthers() {}
    */
    PGPAgent() {
        m_pgPlan = new Plan();
    }

    public boolean accomplishTask(Task task) {
        return true;
    }
    public boolean updateEnvironmentalModel() {
        return true;
    }
    public void exist() {}
    public void act() {}
    public boolean reproduce() {
        return true;
    }
    public void senseEnvironment() {}
    public boolean performAction(Action action) {
        return true;
    }
}

```

```

    }

    public boolean createObject() {return true; }
    public boolean destroyObject() {return true; }
    public boolean perceiveObject() {return true; }
    public boolean modifyObject() {return true; }

}
// end of PGPAgent Class definition

// Plan Class definition
package agent;

import java.util.Vector;
import utils.*;

public class Plan {
    //Vector of Goal objects that this plan achieves or will achieve eventually
    private Vector m_goalsToAchieve;
    private Vector m_actionSequence;

    public Plan() {}
    public Plan(Vector goals){
        m_goalsToAchieve = goals;
    }

    //accessor methods
    public Vector getGoalsToAchieve() {
        return m_goalsToAchieve;
    }
    public void setGoalsToAchieve(Vector goals) {
        m_goalsToAchieve = goals;
    }
    public void setActionSequence(Vector actions) {
        m_actionSequence = actions;
    }
    public Vector getActionSequence() {
        return m_actionSequence;
    }

    public boolean addGoal(Goal newGoal) {
        return m_goalsToAchieve.add(newGoal);
    }
    public boolean removeGoal(Goal goal) {
        return m_goalsToAchieve.remove(goal);
    }
    public boolean addAction(Action action) {
        return m_actionSequence.add(action);
    }
    public boolean removeAction(Action action) {
        return m_actionSequence.remove(action);
    }

    public Plan add(Plan plan) {
        m_actionSequence.addAll(plan.getActionSequence());
        return this;
    }
}
//end of Plan Class definition

```

```

//Planner Class definition
package agent;

import environment.*;
import utils.*;
import java.util.Vector;
import java.util.Stack;
public abstract class Planner {
    private Plan m_currentPlan;
    //Vector of Goal objects
    private Vector m_protectedGoals;
    private WorldState m_initialWorldState;
    //Vector of Action objects
    private Vector m_agentSetOfPossibleActions;
    private Stack m_planGoals;

    //accessor methods
    public Plan getCurrentPlan() {
        return m_currentPlan;
    }
    public void setCurrentPlan(Plan plan) {
        m_currentPlan = plan;
    }
    public Vector getProtectedGoals() {
        return m_protectedGoals;
    }
    public void setProtectedGoals(Vector goals) {
        m_protectedGoals = goals;
    }
    public WorldState getInitialWorldState() {
        return m_initialWorldState;
    }
    public void setInitialWorldState(WorldState worldState) {
        m_initialWorldState = worldState;
    }
    public Vector getAgentSetOfPossibleActions() {
        return m_agentSetOfPossibleActions;
    }
    public void setAgentSetOfPossibleActions(Vector actions) {
        m_agentSetOfPossibleActions = actions;
    }
    public Stack getPlanGoals() {
        return m_planGoals;
    }
    public void setPlanGoals(Stack goals) {
        m_planGoals = goals;
    }

    //virtual methods
    public abstract Plan createPlan(Goal goal);
    public abstract void protectGoal(Goal goal);
    public abstract void resolveConflicts();
    public abstract Action selectAction();
    public abstract boolean checkForConflicts(Action action);
    public abstract boolean isPlanningComplete();
    public abstract void addAction(Action action);
    public abstract void reorderGoals();
}

```

```

public Plan createPlan(Task task) {
    return createPlan(task.getTaskGoal());
}

public Plan createPlan() {
    Stack goals = getPlanGoals();
    if (goals.empty()) {
        return getCurrentPlan();
    } else {
        Goal nextGoal = (Goal)goals.pop();
        return getCurrentPlan().add(createPlan(nextGoal));
    }
}

public void addActionToPlan(Action action) {
    if (checkForConflicts(action)) {
        resolveConflicts();
    }
    addAction(action);
}
}
//end of Planner class definition

```

```

//ReactiveAgent class definition
package agent;

```

```

import utils.*;
import java.util.Set;

```

```

public abstract class ReactiveAgent extends Agent {
    private Set m_setOfRules;

```

```

    public ReactiveAgent() {}
    public ReactiveAgent( Set setOfRules) {
        this();
        m_setOfRules = setOfRules;
    }

```

```

//accessor methods
    public Set getSetOfRules() {
        return m_setOfRules;
    }
    public void setSetOfRules (Set rules) {
        m_setOfRules = rules;
    }

```

```

//virtual methods
    public abstract void senseEnvironment();
    public abstract boolean performAction(Action action);
    public abstract void reactToStimuli();
}
//end of ReactiveAgent class definition

```

```

//SituatedAgent class definition
package agent;

```

```

import java.util.HashMap;

```

```

import utils.*;
public class SituatedAgent extends ReactiveAgent {
    private HashMap m_situatedRules;

    public void addNewRule() {}
    public void removeRule() {}
    public void chooseRule() {}

    //virtual method definitions
    public boolean performAction(Action action) {return true;}
    public void senseEnvironment() {}
    public void reactToStimuli() {}
    public void exist() {}
    public void act() {}
    public boolean reproduce() {return true; }

    public boolean createObject() {return true; }
    public boolean destroyObject() {return true;}
    public boolean perceiveObject() {return true; }
    public boolean modifyObject() {return true; }

}
//end of SituatedAgent class definition

//TaskDrivenAgent class definition

package agent;

import java.util.HashMap;
import java.util.Vector;
import java.util.TreeMap;
import utils.*;

public class TaskDrivenAgent extends CognitiveAgent implements CommunicatingAgent {
    //contains tasks as keys and agents that has send them as values
    private HashMap m_agentTaskMap;
    //contains tasks as a tree following the decomposition process
    private TreeMap m_taskDecompositionTree;
    private boolean m_isOverburdened = false;
    private MultiagentPlanner m_multiagentPlanner;

    public TaskDrivenAgent() {
        super();
        m_agentTaskMap = new HashMap();
        m_taskDecompositionTree = new TreeMap();
    }

    //accessor methods
    public HashMap getAgentTaskMap() {
        return m_agentTaskMap;
    }
    public void setAgentTaskMap(HashMap agentTaskMap) {
        m_agentTaskMap = new HashMap(agentTaskMap);
    }
    public TreeMap getTaskDecompositionTree() {
        return m_taskDecompositionTree;
    }
    public void setTaskDecompositionTree(TreeMap taskTree) {
        m_taskDecompositionTree = new TreeMap(taskTree);
    }
}

```

```

}
public boolean isOverburdened() {
    return m_isOverburdened;
}
public void setOverburdened(boolean overburdened) {
    m_isOverburdened = overburdened;
}

public void addAgentTaskCouple(Agent agent, Task task) {
    m_agentTaskMap.put(agent, task);
}
public MultiagentPlanner getMultiagentPlanner() {
    return m_multiagentPlanner;
}
public void setMultiagentPlanner(MultiagentPlanner planner) {
    m_multiagentPlanner = planner;
}

//returns a vector of Task objects that comprise the original task decomposition
public Vector decomposeTask(Task task) {
    return new Vector();
}
public void reallocateTasks(Vector tasks) {}
public void receiveTask(Task task, Agent agent) {
    m_agentTaskMap.put(task, agent);
    boolean isAccomplished = accomplishTask(task);
}
public boolean composeSolution(Vector tasks, Vector plans) {
    return true;
}
public Vector waitForSolutions(Vector tasks) {
    return new Vector();
}
public void propagateResults(Task task, Vector results) {

    /*
    if the task appears as a key in the agent-task map structure
    the agent corresponding to it can be directly addressed
    otherwise all agents are informed of the results
    */
}

//virtual method definitions
public boolean accomplishTask(Task task) {
    if (isOverburdened()) {
        Vector decomposedTasks = decomposeTask(task);
        reallocateTasks(decomposedTasks);
        Vector results = waitForSolutions(decomposedTasks);
        return composeSolution(decomposedTasks, results);
    } else {
        Planner planner = this.getPlanner();
        Plan plan = planner.createPlan(task);
        Vector results = new Vector();
        boolean success = executePlan(plan, results);
        propagateResults(task, results);
        return success;
    }
}

public boolean updateEnvironmentalModel() {

```

```

        return true;
    }
    public void exist() {}
    public void act() {}
    public boolean reproduce() {return true;}
    public void senseEnvironment() {}
    public boolean performAction(Action action) {return true;}
    public void talkToOthers(Vector agents) {}
    public void listenToOthers() {}

    public boolean createObject() {return true;}
    public boolean destroyObject() {return true;}
    public boolean perceiveObject() {return true;}
    public boolean modifyObject() {return true;}
}
//end of TaskDrivenAgent class definition

```

## *AgentSystem Package*

//AgentSystem class definition

```

package agentsystem;

import environment.*;
import agent.*;
import utils.*;
import java.util.Vector;

public abstract class AgentSystem {
    private String m_systemName;
    private String m_systemType;
    private Environment m_systemEnvironment;
    private Vector m_agentPopulation;
    private Vector m_relations;
    private Vector m_operationSet;
    private MultiagentPlan m_multiagentPlan;
    private PlanningMethod m_planningMethod;

    public AgentSystem() {}
    public AgentSystem(String name, String type, Vector agentPopulation) {
        m_systemName = name;
        m_systemType = type;
        m_agentPopulation = new Vector(agentPopulation);
    }

    //accessor methods
    public String getSystemName() {
        return m_systemName;
    }
    public void setSystemName(String name) {
        m_systemName = name;
    }
    public String getSystemType() {
        return m_systemType;
    }
    public void setSystemType(String type) {
        m_systemType = type;
    }
}

```

```

}
public int getNumOfAgents() {
    return getAgents().size();
}
public Environment getSystemEnvironment() {
    return m_systemEnvironment;
}
public void setSystemEnvironment(Environment env) {
    m_systemEnvironment = env;
}
public Vector getAgents() {
    return m_agentPopulation;
}
public void setAgents(Vector agents) {
    m_agentPopulation = agents;
}
public Vector getRelations() {
    return m_relations;
}
public void setRelations(Vector relations) {
    m_relations = relations;
}
public Vector getOperations() {
    return m_operationSet;
}
public void setOperations(Vector operations) {
    m_operationSet = operations;
}
public MultiagentPlan getMultiagentPlan() {
    return m_multiagentPlan;
}
public void setMultiagentPlan(MultiagentPlan plan) {
    m_multiagentPlan = plan;
}
public PlanningMethod getPlanningMethod() {
    return m_planningMethod;
}
public void setPlanningMethod(PlanningMethod method) {
    m_planningMethod = method;
}

//agent related functions
public boolean addAgent(Agent newAgent) {
    return m_agentPopulation.add(newAgent);
}
public boolean removeAgent(Agent agent) {
    return m_agentPopulation.remove(agent);
}

//relationships related functions

public void addRelationship(Relationship relation) {
    getRelations().add(relation);
}
public void removeRelationship(Relationship relation){
    getRelations().remove(relation);
}
public void changeRelationshipType(Relationship relation, String newType) {
    removeRelationship(relation);
    relation.setRelationshipType(newType);
}

```

```

        addRelationship(relation);
    }
    public void removeObjectFromRelationship(Relationship relation, Object object) {
        relation.removeObjectFromRelationship(object);
    }

    //operations related functions
    public boolean addOperation(Operation newOperation) {
        return m_operationSet.add(newOperation);
    }
    public boolean removeOperation(Operation operation) {
        return m_operationSet.remove(operation);
    }

    public abstract void followPlan(MultiagentPlan plan);
}
//end of AgentSystem class definition

```

```

//CentralizedPlanMergingMethod class definition
package agentsystem;
import agent.*;
import java.util.Vector;
public class CentralizedPlanMergingMethod extends PlanningMethod {
    private CognitiveAgent m_planMerger;

    public CentralizedPlanMergingMethod() {}

    public CognitiveAgent getPlanMerger() {
        return m_planMerger;
    }
    public void setPlanMerger(CognitiveAgent agent) {
        m_planMerger = agent;
    }

    public void gatherAgentPlans() {}
    public void mergePlans() {}
    public MultiagentPlan generatePlan() {
        return new MultiagentPlan();
    }
    public Vector selectPlanningAgents() {
        return new Vector();
    }
}
//end of CentralizedPlanMergingMethod class definition

```

```

//CDDPPlanningMethod class definition

package agentsystem;
import agent.*;
import java.util.Vector;
import java.util.HashMap;
public class CPDPPlanningMethod extends PlanningMethod {
    private TaskDrivenAgent m_centralizedPlannerAgent;

    public CPDPPlanningMethod(String methodName) {
        super(methodName, "centralized");
    }
}

```

```

    m_centralizedPlannerAgent = new TaskDrivenAgent();
}

//accessor methods
public TaskDrivenAgent getCentralizedPlannerAgent() {
    return m_centralizedPlannerAgent;
}
public void setCentralizedPlannerAgent(TaskDrivenAgent agent) {
    m_centralizedPlannerAgent = agent;
}

public Vector selectPlanningAgents() {
    setCentralizedPlannerAgent(chooseAgentToPlan());
    Vector planningAgents = new Vector();
    planningAgents.add(getCentralizedPlannerAgent());
    return planningAgents;
}

public TaskDrivenAgent chooseAgentToPlan() {
    /* should be implemented to select the agent that will
    construct the plan */
    return new TaskDrivenAgent();
}

public MultiagentPlan generatePlan() {
    Vector agents = getParticipatingAgents();
    int nAgentsNum = agents.size();
    for (int i = 0; i < nAgentsNum; i++) {
        TaskDrivenAgent currentAgent = (TaskDrivenAgent)agents.elementAt(i);
        m_centralizedPlannerAgent.addAgentTaskCouple(currentAgent,
currentAgent.getTask());
    }
    return m_centralizedPlannerAgent.getMultiagentPlanner().createMultiagentPlan();
}
}
//end of CDDPPlanningMethod class definition

```

```

//DistributedSystem class definition
package agentsystem;

import java.util.HashMap;
import java.util.Vector;
import utils.*;
public class DistributedSystem extends AgentSystem {
    private HashMap m_taskDistribution;
    private HashMap m_resourceDistribution;
    private HashMap m_skillsDistribution;
    //Vector of Task objects
    private Vector m_problems;
    private Vector m_taskRelations;

    //accessor methods
    public HashMap getTaskDistribution() {
        return m_taskDistribution;
    }
    public void setTaskDistribution(HashMap taskDistribution) {
        m_taskDistribution = taskDistribution;
    }
}

```

```

    }
    public HashMap getResourceDistribution() {
        return m_resourceDistribution;
    }
    public void setResourceDistribution(HashMap resourceDistribution) {
        m_resourceDistribution = resourceDistribution;
    }
    public HashMap getSkillsDistribution() {
        return m_skillsDistribution;
    }
    public void setSkillsDistribution(HashMap skillsDistribution) {
        m_skillsDistribution = skillsDistribution;
    }
    public Vector getDistributedProblems() {
        return m_problems;
    }
    public void setDistributedProblems(Vector problems) {
        m_problems = problems;
    }
    public Vector getTaskRelations() {
        return m_taskRelations;
    }
    public void setTaskRelations(Vector relations) {
        m_taskRelations = relations;
    }
    public void addTaskRelation(Relationship relation) {
        if (relation instanceof TaskRelationship)
            getTaskRelations().add(relation);
    }

    public void followPlan(MultiagentPlan plan) {}
}
//end of DistributedSystem class definition

```

//MultiagentPlan class definition

```

package agentsystem;
import java.util.HashMap;
import agent.*;
public class MultiagentPlan {
    //maps each agent to the plan it has to follow
    HashMap m_agentPlanMap;

    public MultiagentPlan() {
        m_agentPlanMap = new HashMap();
    }

    //accessor methods
    public HashMap getAgentPlanMapping() {
        return m_agentPlanMap;
    }
    public void setAgentPlanMapping(HashMap agentPlanMap) {
        m_agentPlanMap = agentPlanMap;
    }

    public Plan getPlanForAgent(Agent agent) {
        if (m_agentPlanMap.containsKey(agent)) {
            return (Plan)m_agentPlanMap.get(agent);
        }
    }
}

```

```

    }
    return new Plan();
}

public void savePlanForAgent(Agent agent, Plan plan) {
    m_agentPlanMap.put(agent,plan);
}

}
//end of MultiagentPlan class definition

```

```

//MultiagentSystem class definition
package agentsystem;
import agent.*;
public class MultiagentSystem extends AgentSystem {
    public void registerAgent(Agent agent) {}
    public void unregisterAgent(Agent agent) {}
    public void followPlan(MultiagentPlan plan) {}
}
//end of MultiagentSystem class definition

```

```

//PlanningMethod class definition
package agentsystem;

import java.util.Vector;
public abstract class PlanningMethod {
    private String m_methodName;
    private String m_methodType;
    private Vector m_planningAgents;
    private Vector m_participatingAgents;

    PlanningMethod() {}
    PlanningMethod(String name, String type) {
        m_methodName = name;
        m_methodType = type;
        m_planningAgents = new Vector();
        m_participatingAgents = new Vector();
    }

    //accessor methods
    public String getMethodName() {
        return m_methodName;
    }
    public void setMethodName(String name) {
        m_methodName = name;
    }
    public void setMethodType(String type) {
        m_methodType = type;
    }
    public String getMethodType() {
        return m_methodType;
    }
    public Vector getPlanningAgents() {
        return m_planningAgents;
    }
    public void setPlanningAgents(Vector agents) {
        m_planningAgents = new Vector(agents);
    }
}

```

```

public Vector getParticipatingAgents() {
    return m_participatingAgents;
}
public void setParticipatingAgents(Vector agents) {
    m_participatingAgents = new Vector(agents);
}

//virtual methods
public abstract Vector selectPlanningAgents();
public abstract MultiagentPlan generatePlan();
}
//end of PlanningMethod class definition

```

## *Environment Package*

```

//AccessibleEnvironment interface definition
package environment;
import agent.*;

public interface AccessibleEnvironment {
    public EnvironmentModel getCompleteEnvironment(Agent agent);
}
//end of AccessibleEnvironment interface definition

//ContinuousEnvironment interface definition

package environment;
import agent.*;
import java.util.Vector;

public interface ContinuousEnvironment {
    public Vector getAgentLocalPercepts(Agent agent, WorldState worldState);
}
//end of ContinuousEnvironment interface definition

//DeterministicEnvironment interface definition
package environment;

public interface DeterministicEnvironment {

    public WorldState getActionResult();

}
//end of DeterministicEnvironment interface definition

//DiscreteEnvironment interface definition

package environment;

import utils.*;
import agent.*;
import java.util.Vector;

public interface DiscreteEnvironment {

```

```

    public Vector enumerateActions();
    public Vector enumeratePercepts();
    public Vector enumerateAgentPossibleActions(Agent agent);
    public Vector getAgentPercepts(Agent agent, WorldState worldState);
}
//end of DiscreteEnvironment interface definition

//DynamicEnvironment interface definition

package environment;

public interface DynamicEnvironment {
    public WorldState evolve();
    WorldState changeEnvironment(Action action);
}
//end of DynamicEnvironment interface definition

//Environment class definition

package environment;

import java.util.Vector;
import agent.*;
import environment.*;
import utils.*;

public abstract class Environment {

    private WorldState m_currentState;
    //Vector containing the objects in the World that differ from agents
    private Vector m_objects;
    private int m_dimensions;

    public Environment() {}

    //accessor methods
    public void setWorldState(WorldState worldState) {
        m_currentState = worldState;
    }
    public WorldState getWorldState() {
        return m_currentState;
    }
    public void setWorldObjects(Vector objects) {
        m_objects = objects;
    }
    public Vector getWorldObjects() {
        return m_objects;
    }
    public int getWorldDimensions() {
        return m_dimensions;
    }
    public void setWorldDimensions(int dimensions) {
        m_dimensions = dimensions;
    }
}

public boolean addWorldObject(Object obj) {
    return m_objects.add(obj);
}

```

```

    }
    public boolean removeWorldObject(Object obj) {
        return m_objects.remove(obj);
    }
    public boolean updateWorldObject(Object obj) {
        return true;
    }
    public boolean manipulateObject(Agent agent, Object object) {
        return true;
    }
    public boolean manipulateObjects(Vector agents, Vector objects) {
        return true;
    }
    public void setObjectPosition(Object obj, Position pos) { }
    public void changeWorldState(Action action) {}
    // public Position getObjectPosition(Object obj){}
}

```

```

//InaccessibleEnvironment interface definition
package environment;

```

```

import agent.*;
public interface InaccessibleEnvironment {
    public EnvironmentModel getLocalEnvironment();
}
//end of InaccessibleEnvironment interface definition

```

```

//NonDeterministicEnvironment interface definition

```

```

package environment;

```

```

import utils.*;
import java.util.Vector;

```

```

public interface NonDeterministicEnvironment {
    //To return vector of WorldState (ProbableWorldState) objects
    public Vector getPossibleResults(Action action);
}
//end of NonDeterministicEnvironment interface definition

```

```

//Position class definition

```

```

package environment;

```

```

public class Position {
}
//end of Position class definition

```

```

//StaticEnvironment interface definition

```

```

package environment;

```

```

import utils.*;

```

```

public interface StaticEnvironment {

```

```

        WorldState changeEnvironment(Action action);
    }
//end of StaticEnvironment interface definition

//STRIPSWorldState class definition

package environment;

import java.util.Vector;
public class STRIPSWorldState extends WorldState {
    private Vector m_literals;

    STRIPSWorldState(Vector literals) {
        super();
        m_literals = literals;
    }
    //accessor methods
    public Vector getLiterals() {
        return m_literals;
    }
    public void setLiterals(Vector literals) {
        m_literals = literals;
    }
    public void addLiterals(Vector literals) {
        m_literals.addAll(literals);
    }
    public void removeLiterals(Vector literals) {
        m_literals.removeAll(literals);
    }
}
//end of STRIPSWorldState class definition

//WorldState class definition

package environment;

import java.util.Vector;
import utils.*;

public class WorldState {
    private long m_time;
    private Vector m_currentStateRel;
    private Vector m_currentStateStimuli;

    public WorldState() {}
    public WorldState(long time, Vector initialState, Vector initialStimuli) {
        m_time = time;
        m_currentStateRel = initialState;
        m_currentStateStimuli = initialStimuli;
    }

    //accessor methods
    public long getTime() {
        return m_time;
    }
    public void setTime(long time) {

```

```

    m_time = time;
}
public Vector getCurrentStateRelations() {
    return m_currentStateRel;
}
public void setCurrentStateRelations(Vector relations) {
    m_currentStateRel = relations;
}
public Vector getCurrentStateStimuli() {
    return m_currentStateStimuli;
}
public void setCurrentStateStimuli(Vector stimuli) {
    m_currentStateStimuli = stimuli;
}

public void initWorldState() {}
public void initWorldState (Vector initialState) {}
public void changeWorldState(Action action) {
    action.execute(this);
}
//maybe states should be created instead of changed so that history is tracked?
}
//end of WorldState class definition

```

## *Utils Package*

```

//Action class definition

package utils;

import java.util.Vector;
import agentsystem.*;
import environment.*;

public abstract class Action {
    private String m_actionName;
    //a Vector of Resource objects that the action will consume
    private Vector m_resources;

    public Action() {
        m_actionName = "";
        m_resources = new Vector();
    }
    public Action(String actionName) {
        m_actionName = actionName;
        m_resources = new Vector();
    }
    public Action(String actionName, Vector resources) {
        this(actionName);
        m_resources = new Vector(resources);
    }

    //accessor methods
    public String getActionName() {
        return m_actionName;
    }
    public void setActionName(String actionName) {

```

```

        m_actionName = actionName;
    }
    public Vector getResources() {
        return m_resources;
    }
    public void setResources(Vector resources) {
        m_resources = new Vector(resources);
    }

    public abstract void execute(WorldState worldState);
    public boolean isActionApplicable(WorldState worldState) {
        return true;
    }
}
//end of Action class definition

//ConjunctiveGoal class definition
package utils;

import java.util.Vector;

public class ConjunctiveGoal extends Goal {
    //Vector containing the Literal objects that comprise this goal
    private Vector m_literals;

    public ConjunctiveGoal(Vector literals) {
        super();
        m_literals = literals;
    }

    public ConjunctiveGoal(Vector literals, boolean bProtected) {
        super(bProtected);
        m_literals = literals;
    }

    public Vector getLiterals() {
        return m_literals;
    }
    public void setLiterals(Vector literals) {
        m_literals = literals;
    }

    public void setGoalProtection(boolean bProtected) {
        super.setGoalProtection(bProtected);
        int nNumofLiterals = m_literals.size();
        for (int i=0; i<nNumofLiterals; i++) {
            Literal literal = (Literal)m_literals.elementAt(i);
            literal.setProtected(bProtected);
        }
    }
}
//end of ConjunctiveGoal class definition

//ConsumeOperation class definition
package utils;
import agent.*;

```

```

import java.util.Vector;
public class ConsumeOperation extends Operation {
    public boolean perform(Agent agent, Object subject) {
        return true;
    }
    public boolean perform(Vector agents, Vector subjects) {
        return true;
    }
}

//EnableRelationship class definition
package utils;
import java.util.Vector;
public class EnableRelationship extends TaskRelationship {
    public EnableRelationship(Task enabler, Vector tasks) {
        super("enable", enabler, tasks);
    }
}
//end of EnableRelationship class definition

//Event class definition
package utils;
import java.util.Vector;
import environment.*;

public class Event extends Action {
    // Vector of condition literals that cause the event to happen
    private Vector m_triggers;

    public Event() {}
    public Event(String eventName) {
        super(eventName);
        m_triggers = new Vector();
    }

    public void setTriggers(Vector triggers) {
        m_triggers = triggers;
    }
    public Vector getTriggers() {
        return m_triggers;
    }
    public boolean addTrigger (Literal trigger) {
        return m_triggers.add(trigger);
    }
    public boolean removeTrigger(Literal trigger) {
        return m_triggers.remove(trigger);
    }
    public void triggerEvent(WorldState worldState) {}
    public void execute(WorldState worldState) {
        triggerEvent(worldState);
    }
}
//end of Event class definition

//FacilitateRelationship class definition
package utils;
import java.util.Vector;

```

```

import utils.*;

public class FacilitateRelationship extends TaskRelationship {
    public FacilitateRelationship(Task facilitator, Vector tasks) {
        super("facilitate", facilitator, tasks);
    }
}
//end of FacilitateRelationship class definition

//Goal class definition

package utils;

import java.util.Vector;

public class Goal {
    private Vector m_goalRelationships;
    private boolean m_isGoalProtected = false;

    public Goal() {
        m_goalRelationships = new Vector();
    }

    public Goal(boolean bProtection) {
        this();
        m_isGoalProtected = bProtection;
    }

    //accessor methods
    public Vector getGoalRelationships() {
        return m_goalRelationships;
    }
    public void setGoalRelationships(Vector goalRelationships) {
        m_goalRelationships = goalRelationships;
    }
    public boolean isGoalProtected() {
        return m_isGoalProtected;
    }
    public void setGoalProtection(boolean bProtected) {
        m_isGoalProtected = bProtected;
    }
}
//end of Goal class definition

//HinderRelationship class definition

package utils;
import java.util.Vector;
public class HinderRelationship extends TaskRelationship {
    public HinderRelationship(Task hinderer, Vector tasks) {
        super("hinder", hinderer, tasks);
    }
}
//end of HinderRelationship class definition

//InhibitRelationship class definition

```

```

package utils;
import java.util.Vector;
import utils.*;
public class InhibitRelationship extends TaskRelationship {
    public InhibitRelationship(Task inhibitor, Vector tasks) {
        super("inhibit", inhibitor, tasks);
    }
}
//end of InhibitRelationship class definition

//Literal class definition

package utils;

import java.util.Vector;

public class Literal {
    private String m_name;
    private Object m_object;
    private Vector m_subjects;
    private boolean m_protected;

    public Literal() {}
    public Literal(String name, Object object, Vector subjects) {
        m_name = name;
        m_object = object;
        m_subjects = subjects;
    }

    //accessor methods
    public void setName(String name) {
        m_name = name;
    }
    public String getName() {
        return m_name;
    }
    public void setObject(Object object) {
        m_object = object;
    }
    public Object getObject() {
        return m_object;
    }
    public Vector getSubjects() {
        return m_subjects;
    }
    public void setSubjects(Vector subjects) {
        m_subjects = subjects;
    }
    public void setProtected(boolean protection) {
        m_protected = protection;
    }
    public boolean isProtected() {
        return m_protected;
    }
}
//end of Literal class definition

```

```

//MacroAction class definition

package utils;

import java.util.Vector;

public abstract class MacroAction extends Action {
    //Vector of Action objects
    private Vector m_actionSequence;

    public MacroAction() {}
    public MacroAction(Vector actionSequence) {
        m_actionSequence = actionSequence;
    }

    public Vector getActionSequence() {
        return m_actionSequence;
    }
    public abstract boolean execute();
}
//end of MacroAction class definition

//ManipulateOperation class definition
package utils;
import agent.*;
import java.util.Vector;

public class ManipulateOperation extends Operation {
    public boolean perform(Agent agent, Object subject) {return true;}
    public boolean perform(Vector agents, Vector subjects) {return true;}
}
//end of ManipulateOperation class definition

//Operation class definition
package utils;

import java.util.Vector;
import agent.*;

public abstract class Operation {
    private String m_name;

    //accessor methods
    public String getOperationName() {
        return m_name;
    }
    public void setOperationName(String name) {
        m_name = name;
    }

    public abstract boolean perform(Agent agent, Object subject);
    public boolean perform(Vector agents, Vector subjects) {
        return true;
    }
}
//end of Operation class definition

```

```

//PerceiveOperation class definition
package utils;
import agent.*;

public class PerceiveOperation extends Operation {
    public boolean perform(Agent agent, Object subject) {
        return true;
    }
}
//end of Perceive class definition

//Perception class definition
package utils;

public class Perception {
}
//end of Perception class definition

//ProduceOperation class definition
package utils;
import agent.*;
import java.util.Vector;

public class ProduceOperation extends Operation {
    public boolean perform(Agent agent, Object subject) {
        return true;
    }
    public boolean perform(Vector agents, Vector objects) {
        return true;
    }
}
//end of ProduceOperation class definition

//Relationship class definition
package utils;
import java.util.Vector;

public abstract class Relationship {
    private String m_relType;
    private String m_relName;
    private boolean m_isHoldRelationship;
    private Vector m_objects;

    public Relationship() {}
    public Relationship(String relType, String relName) {
        m_relType = relType;
        m_relName = relName;
    }
    public Relationship(String relType, String relName, Vector objects) {
        this(relType, relName);
        m_objects = new Vector(objects);
    }
}

public Relationship(String relType, String relName,
                    boolean holdRelation, Vector objects){
    this(relType, relName, objects);
}

```

```

    m_isHoldRelationship = holdRelation;
}

//accessor methods
public String getRelationshipType() {
    return m_relType;
}
public void setRelationshipType(String relType) {
    m_relType = relType;
}
public String getRelationshipName() {
    return m_relName;
}
public void setRelationshipName(String relName) {
    m_relName = relName;
}
public Vector getRelationshipObjects() {
    return m_objects;
}
public void setRelationshipObjects(Vector objects) {
    m_objects = objects;
}
public boolean isRelationshipHolding() {
    return m_isHoldRelationship;
}
public void setRelationshipHolding(boolean hold) {
    m_isHoldRelationship = hold;
}

public abstract void addObjectToRelationship(Object object);
public abstract void removeObjectFromRelationship(Object object);

}
//end of Relationship class definition

//Requirements class definition
package utils;

import java.util.Vector;
public class Requirements {
    //Vector of TimeInterval objects specifying at what periods the task shpuld be performed?
    private Vector m_timeLimitations;
    //Vector of Resource objects specifying the resources that can be used when performing
the task
    private Vector m_resources;
    //a String description of other requirements
    private String m_otherRequirements;

    Requirements() {}

    //accessor methods
    public Vector getTimeLimitations() {
        return m_timeLimitations;
    }
    public void setTimeLimitations(Vector limitations) {
        m_timeLimitations = limitations;
    }
    public Vector getResources() {

```

```

        return m_resources;
    }
    public void setResources(Vector resources) {
        m_resources = resources;
    }
    public String getOtherRequirements() {
        return m_otherRequirements;
    }
    public void setOtherRequirements(String requirements) {
        m_otherRequirements = requirements;
    }
}
//end of Requirements class definition

```

```

//Resource interface definition
package utils;

```

```

public interface Resource {
    public Object getCapacity();
    public boolean reserve();
    public boolean free();
    public void consume();
}
//end of Resource interface definition

```

```

//Result interface definition

```

```

package utils;

```

```

public interface Result {
    public Object getContent();
    public float getCredibilityFactor();
}
//end of Result interface definition

```

```

//STRIPSAction class definition

```

```

package utils;

```

```

import java.util.Vector;
import environment.*;
public class STRIPSAction extends Action {
    //a Vector of Literals
    private Vector m_preconditions;
    //a Vector of Literals
    private Vector m_postConditions;
    //an optional Vector of Literals
    private Vector m_duringConditions;

    public STRIPSAction() {
        super();
        m_preconditions = new Vector();
        m_postConditions = new Vector();
        m_duringConditions = new Vector();
    }
}

```

```

public STRIPSAction(String name, Vector preconditions, Vector postconditions) {

```

```

    super(name);
    m_preconditions = new Vector(preconditions);
    m_postConditions = new Vector(postconditions);
    m_duringConditions = new Vector();
}

public STRIPSAction(String name, Vector preconditions,
                    Vector postconditions, Vector resources) {
    this(name, preconditions, postconditions);
    setResources(resources);
}

public Vector getPreconditions() {
    return m_preconditions;
}

public void setPreconditions(Vector preconditions) {
    m_preconditions = new Vector(preconditions);
}

public Vector getPostconditions() {
    return m_postConditions;
}

public void setPostconditions(Vector postconditions) {
    m_postConditions = new Vector(postconditions);
}

public Vector getDuringConditions() {
    return m_duringConditions;
}

public void setDuringConditions(Vector duringConditions) {
    m_duringConditions = new Vector(duringConditions);
}

public boolean isActionApplicable(WorldState worldState) {
    boolean bApplicable = true;
    int nPostCondSize = getPostconditions().size();
    Vector literals = ((STRIPSWorldState)worldState).getLiterals();
    for (int i = 0; i <nPostCondSize; i++) {
        bApplicable = literals.contains(getPostconditions().elementAt(i));
        if (!bApplicable) return bApplicable;
    }
    return bApplicable;
}

public void execute(WorldState worldState) {
    if (isActionApplicable(worldState)) {
        ((STRIPSWorldState)worldState).addLiterals(getPostconditions());
    }
}

}
//end of STRIPSAction class definition

//SystemObject class definition
package utils;

public abstract class SystemObject {
    private String m_objectName;
    private String m_objectType;

    public SystemObject() {}

```

```

public SystemObject(String name, String type) {
    m_objectName = name;
    m_objectType = type;
}

//accessor methods
public String getName() {
    return m_objectName;
}
public void setName(String name) {
    m_objectName = name;
}
public String getType() {
    return m_objectType;
}
public void setType(String type) {
    m_objectType = type;
}

public abstract boolean createObject();
public abstract boolean modifyObject();
}
//end of SystemObject class definition

//Task class definition

package utils;

import agent.*;
import environment.*;
import java.util.Vector;

public class Task {
    //the requirements that have to be met by the task solution or the process of solving it
    private Requirements m_requirements;
    //Vector of objects that will be manipulated while performing the task
    private Vector m_objects;
    // vector of the Action objects that represent actions that can be executed while completing
the task
    private Vector m_possibleActions;
    // the task goal
    private Goal m_goal;
    private WorldState m_initialWorldState;
    private Vector m_staticObjects;
    //Vector of Objects that remain static during the task but have to be considered for its
completion
    private Environment m_environment;

    public Task() {}

    //accessor methods
    public Requirements getRequirements() {
        return m_requirements;
    }
    public void setRequirements(Requirements requirements) {
        m_requirements = requirements;
    }
    public Vector getTaskObjects() {

```

```

    return m_objects;
}
public void setTaskObjects(Vector objects) {
    m_objects = objects;
}
public Vector getPossibleActions() {
    return m_possibleActions;
}
public void setPossibleActions(Vector actions) {
    m_possibleActions = actions;
}
public Goal getTaskGoal() {
    return m_goal;
}
public void setTaskGoal(Goal goal) {
    m_goal = goal;
}
public WorldState getInitialWorldState() {
    return m_initialWorldState;
}
public void setInitialWorldState(WorldState worldState) {
    m_initialWorldState = worldState;
}
public Vector getStaticObjects() {
    return m_staticObjects;
}
public void setStaticObjects(Vector staticObjects) {
    m_staticObjects = staticObjects;
}
public Environment getEnvironment() {
    return m_environment;
}
public void setEnvironment(Environment environment) {
    m_environment = environment;
}
}
//end of Task class definition

```

```

//TaskRelationship class definition
package utils;

```

```

import java.util.Vector;

```

```

public class TaskRelationship extends Relationship{
    private Task m_basicTask;
    private Vector m_relatedTasks;

    public TaskRelationship(String relName, Task basicTask, Vector relatedTasks) {
        super("task relationship", relName);
        Vector objects = new Vector();
        objects.add(basicTask);
        objects.addAll(relatedTasks);
        super.setRelationshipObjects(objects);
    }

    public Task getBaseTask() {
        return m_basicTask;
    }
    public Vector getRelatedTasks() {

```

```

        return m_relatedTasks;
    }
    public void addObjectToRelationship(Object object) {
        getRelationshipObjects().add(object);
    }
    public void removeObjectFromRelationship(Object object) {
        Vector relationshipObjects = getRelationshipObjects();
        if (!object.equals(relationshipObjects.firstElement())) {
            relationshipObjects.remove(object);
        }
    }
}
//end of TaskRelationship class definition

```

//TransformOperation class definition

```

package utils;
import agent.*;
import java.util.Vector;

public class TransformOperation extends Operation {
    public boolean perform(Agent agent, Object subject) {
        return true;
    }
    public boolean perform(Vector agents, Vector subjects) {
        return true;
    }
}
//end of TransformOperation class definition

```

